Technical Report: Designing High-Performance Real-Time SDRAM Controllers for Many-Core Systems (Revision 1.0)

Leonardo Ecco and Rolf Ernst

May 3, 2017

Author: Leonardo Ecco and Rolf Ernst
{ecco,ernst}@tu-bs.de
Technische Universitaet Braunschweig
Institute fuer Datentechnik und Kommunikationsnetze
Hans-Sommer-Str. 66
38106 Braunschweig
Germany

Contents

1 Introduction

DDR SDRAMs (*Double Data Rate Synchronous Dynamic Random Access Memories*) represent high-density and low-cost storage devices and, hence, are widely employed in multi- and many-core platforms, e.g. [1] and [2]. However, the aforementioned benefits come at the price of a *stateful* two-stage access protocol, which reflects the bank-based structure and an internal level of explicitly managed caching present in these devices. In a scenario in which multiple requestors share a SDRAM, the *stateful* nature of the protocol poses a predictability challenge. To overcome such challenge, several real-time SDRAM controllers have been proposed. They can be classified into two main groups: *close-row* controllers, which traditionally rely on *pattern-based* bank interleaving and only exploit caching locality within the boundary of a single request [3, 4, 5], and *open-row* controllers, which potentially exploit caching locality over the boundary of several requests, [6, 7, 8, 9].

Both approaches have drawbacks and advantages and a comparison between them is not the focus of this technical report. In this report, we limit ourselves to pointing out that traditional *close-row* controllers are highly effective in scenarios in which the ratio between request granularity and SDRAM data bus width is large [10], e.g. large request and narrow data bus. However, in many-core platforms like [1] or [2], in which processors mainly retrieve cache lines from (up to 4) 64-bit wide SDRAM modules, such ratio is small. For such scenario, researchers proposed *open-row* real-time controllers.

Unlike the case for *close-row* controllers [11], the evaluation of *open-row* SDRAM controllers for the real-time domain has been mostly generation-specific. This poses an interesting challenge because each DDR generation differs from the previous one in architectural features and/or timing constraints. For instance, DDR4 SDRAMs introduced *bank groups*, which in turn created new timing constraints. Hence, in this technical report, we **extend** our previous work [8, 9] in order to address such challenge.

More specifically, this technical report provides the following **main contributions**:

1. We propose a multi-generation *open-row* real-time SDRAM controller architecture. Our architecture implements read/write bundling [8, 9] in order to minimize data bus turnarounds and rank switching events, which are better discussed in Sections 2.2 and 2.3.

2. We provide a detailed timing analysis of our architecture. Moreover, in order to keep the analysis generation-independent, we propose a generic and flexible notation to represent SDRAM timing constraints. Such notation can be employed by other work in the area of SDRAM controllers.

3. Our architecture and timing analysis cover the DDR4 standard. To our knowledge, this is the first technical report to consider DDR4 SDRAMs in the context of *open-row* real-time SDRAM controllers.

4. We examine the trends in terms of worst-case performance over different speed bins and module configurations from DDR2, DDR3 and DDR4 SDRAMs.

The rest of this report is organized as follows: In Section 2, we provide the background on SDRAM systems and discuss the related work. In Section 3, we propose a flexible notation to represent SDRAM timing constraints. In Section 4, we describe the architecture of our multi-generation *open-row* real-time SDRAM controller. In Section 5, we provide a detailed timing

analysis of our SDRAM controller architecture. Finally, in Section 6, we present an evaluation of our approach, followed by our concluding remarks in Section 7.

2 Background

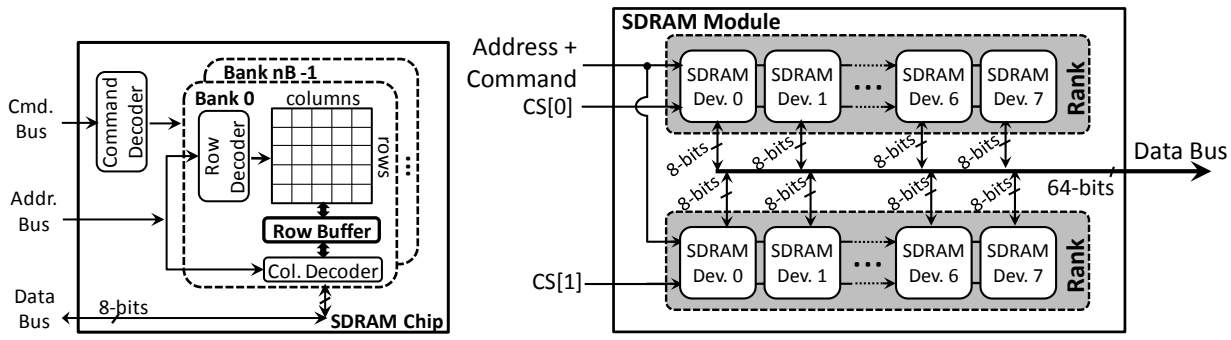
In this section, we provide the background on SDRAM systems. Moreover, we also provide a discussion about real-time SDRAM controllers.

2.1 SDRAM Naming Conventions

DDR SDRAMs are identified by a string that uses the following pattern: *DDRx-(speed bin)(grade)*. The x stands for the generation, e.g. DDR2 or DDR3. The speed bin is represented using the theoretical peak data rate measured in MT/s (mega transfers per second), which corresponds to 2 times the frequency of the data bus measured in MHz (because of the double data rate). For instance, a DDR3-800E device is able to perform up to 800 MT/s and its data bus frequency is equal to 400 MHz. The letter appended to the end of the string distinguishes between devices from the same speed bin that have different timing constraints (the closer to 'A' the grade is, the smaller the constraints).

2.2 Devices, Commands and Timing Constraints

We depict the logical structure of a SDRAM device in Fig. 2.1a. A SDRAM device is divided into banks. The exact number of banks in a SDRAM device varies across different generations, with possible values being 4 or 8 for DDR2, 8 for DDR3, and 8 or 16 for DDR4. For DDR4, the banks are further divided into *bank groups* of 4 banks. The role played by *groups* on command scheduling is discussed later.



(a) SDRAM device.

(b) Dual-rank SDRAM module.

Figure 2.1: SDRAM system.

Each bank contains a matrix-like structure and a row buffer. The matrix-like structures are not visible to the SDRAM controller. All data exchanges are instead performed through the corresponding row buffer, which represents the internal level of caching mentioned in the introduction. There are four commands used to move data into/from a row buffer: *activate* (A), *precharge* (P), *read* (R) and *write* (W). The *activate* command loads a matrix row into the corresponding row buffer, which is known as *opening a row*. The *precharge* command writes the contents of a row buffer back into the corresponding matrix, which is known as *closing a row*. The *read* and *write* commands are used to retrieve or forward words from or into a row buffer. We use the acronym CAS (Column Address Strobe) to refer to both *read* and *write* commands and we use the letter

(C) to refer to a CAS command.

CAS commands operate in bursts, which means that each of them transfers more than one word. The exact amount of words transferred by a CAS command is determined by the the burst length (BL) parameter. A burst length of 8 words is supported by all DDR families investigated in this technical report. A single CAS command occupies the data bus for $t_{BURST} = BL/2 = 4$ cycles and transfers $BL \cdot W_{BUS}$ bits, where W_{BUS} represents the width of the data bus. In this technical report, we consider systems with $BL = 8$ and $W_{BUS} = 64$ bits, in which a single CAS command transfers 64 bytes (a common cache line size).

Table 2.1: Minimum timing interval between cmd_a and cmd_b for three different generations of DDRx SDRAM. Extracted from [12], [13] and [14].

SDRAM Gen.	cmd_a	cmd_b=P Same bank	cmd_b=P Diff. bank	cmd_b=A Same Bank	cmd_b=A Diff. bank	cmd_b=R Same bank	cmd_b=R Diff. bank	cmd_b=W Same Bank	cmd_b=W Diff. bank
DDR2	A	t_{RAS}	n.a.	t_{RC}	t_{RRD}	t_{RCD}	n.a.	t_{RCD}	n.a.
DDR2	P	n.a.	1	t_{RP}	n.a.	n.a.	n.a.	n.a.	n.a.
DDR2	R	$t_{BURST} - 2 + \max(t_{RTP}, 2)$	n.a.	n.a.	n.a.	t_{CCD}		4 if $t_{BURST} = 2$, else 6	
DDR2	W	$t_{BURST} + t_{WL} + t_{WR}$	n.a.	n.a.	n.a.	$t_{RL} - 1 + t_{BURST} + t_{WTR}$		t_{CCD}	
DDR3	A	t_{RAS}	n.a.	t_{RC}	t_{RRD}	t_{RCD}	n.a.	t_{RCD}	n.a.
DDR3	P	n.a.	1	t_{RP}	n.a.	n.a.	n.a.	n.a.	n.a.
DDR3	R	$\max(t_{RTP}, 4)$	n.a.	n.a.	n.a.	t_{CCD}		$t_{RL} + t_{BURST} + 2 - t_{WL}$	
DDR3	W	$t_{BURST} + t_{WL} + t_{WR}$	n.a.	n.a.	n.a.	$t_{WL} + t_{BURST} + t_{WTR}$		t_{CCD}	
DDR4	A	t_{RAS}	n.a.	t_{RC}	t_{RRD_x}	t_{RCD}	n.a.	t_{RCD}	n.a.
DDR4	P	n.a.	1	t_{RP}	n.a.	n.a.	n.a.	n.a.	n.a.
DDR4	R	t_{RTP}	n.a.	n.a.	n.a.	t_{CCD_x}		$t_{RL} + t_{BURST} - t_{WL} + t_{PREAMBLE}$	
DDR4	W	$t_{BURST} + t_{WL} + t_{WR}$	n.a.	n.a.	n.a.	$t_{WL} + t_{BURST} + t_{WTR\_x}$		t_{CCD_x}	

We discuss SDRAM timing constraints. The documents that specify the DDR2/3/4 standards [12, 13, 14] describe in detail how each command changes the state of a SDRAM device and the time interval required for such changes to be performed. However, as described in [11], from the perspective of the SDRAM controller, those details can be abstracted into timing constraints that dictate a minimum distance between consecutive SDRAM commands. We enumerate such constraints for DDR2/3/4 in Table 2.1 and discuss them in detail.

In the table, notice that several cells are marked as *not applicable* (n.a.). Those refer to command combinations that are either invalid or unconstrained. For instance, if cmd_a is a *write* then cmd_b cannot be an *activate* (in the same bank), because the corresponding row buffer would have to be firstly precharged. The other values present in the table cells refer to labels of the timing intervals required for command-triggered changes in a SDRAM device to complete. Except for t_{BURST}, which was defined by us (as the duration of a data transfer), all other labels are employed in the DDR2/3/4 specifications[1].

For DDR4 devices, notice that some of the timing interval labels have a x suffix, which indicates that the value depends on whether cmd_a and cmd_b target banks that belong to the same *bank group* or not. If that is the case, the intervals are longer. In the DDR4 specification, the difference is identified by suffixing the corresponding labels with L (from long) or S (from short). For instance, consider two consecutive *activate* commands to different banks. If the banks are part of the same *bank group*, then the commands must be executed at least t_{RRD_L} cycles apart. However, if the banks are not part of the same *bank group*, they must be at least t_{RRD_S} cycles apart. Moreover, $t_{RRD\_L} > t_{RRD\_S}$.

Also to be highlighted is the fact that some constraints are the same regardless of whether cmd_a and cmd_b target the same bank or not. This is the case for any two consecutive CAS commands, which include not only the minimum distance between two *writes* or two *reads*, but also the data

[1] The DDR2/3/4 specifications employ the acronyms WL (write latency) and CWL (CAS write latency) interchangeably. The same observations apply to RL (read latency) and CL (CAS latency). In Table 2.1, we stick to the WL and RL acronyms.

bus turnarounds mentioned in the introduction (cells in which $cmd_a = W$ and $cmd_b = R$ or vice-versa). As detailed in [8], the faster the SDRAM device, the larger the overhead for data bus turnarounds. Consequently, the controller proposed in this technical report reorders CAS commands in order to minimize such overhead.

Finally, we highlight that SDRAMs have a fifth command that is not related to data transfers: the *refresh* (R). SDRAMs must be refreshed every $t_{REFI} = 7.8\mu s$ in order to prevent the capacitors that store data from being discharged. The amount of cycles required for a REF command to complete (referred to as t_{RFC}) varies according to the SDRAM device, e.g. $t_{RFC} = 36$ cycles for a DDR3-800E.

2.3 Multi-Rank Modules

Individual SDRAM devices have data buses of 4, 8 or 16 bits. However, they are usually grouped under the same clock and chip-select signal in a so called *rank*, thus forming a virtual device with a larger data bus and storage capacity. For the sake of clarity and ease of comprehension, we depict a generic dual-rank SDRAM module in Fig. 2.1b.

Notice that even though each rank has its own chip-select signal, both ranks share the same command, address and data buses (the clock is also shared, although not depicted in the figure). All these signals and buses are driven exclusively by the SDRAM controller, except for the data bus, which has multiple drivers. For a write operation, the SDRAM controller drives the data bus, while for a read, the corresponding rank does it (in order to send data back to the controller). Because the data bus has multiple drivers, i.e. it is a multi-drop bus, there must be and idle timing interval between consecutive data transfers initiated by different senders so that the integrity of the electrical signals being transmitted is enforced. This timing interval is known as the *rank-to-rank switching time*. From now on, we refer to it simply as t_{RTRS}.

From the perspective of command generation, it is useful to think of t_{RTRS} in terms of the impact it has on the minimum distances between consecutive CAS commands executed in different ranks. More specifically, as detailed in [9], t_{RTRS} leads to the so-called *inter-rank* timing constraints enumerated in Table 2.2. For ease of understanding, we provide a graphical depiction of such inter-rank timing constraints in Fig. 2.2.

Table 2.2: Inter-rank timing constraints.

SDRAM	cmd_a	$cmd_b = R$ (executed in diff. rank)		$cmd_b = W$ (executed in diff. rank)	
Gen.		Notation used in [9]	Computed as	Notation used in [9]	Computed as
DDR2/3/4	R	t_{RDRD_dr}	$t_{BURST} + t_{RTRS}$	t_{RDWR_dr}	$t_{RL} - t_{WL} + t_{BURST} + t_{RTRS}$
	W	t_{WRRD_dr}	$t_{WL} - t_{RL} + t_{BURST} + t_{RTRS}$	t_{WRWR_dr}	t_{BURST}

Finally, we highlight that t_{RTRS} is obtained experimentally. For our evaluation, we assume a t_{RTRS} of 4.5 nano seconds for a dual-rank module, as it was reported in [15]. Moreover, we also highlight that SDRAM controllers measure time in terms of data bus clock cycles. Consequently, in terms of cycles, the rank switching overhead is larger for faster SDRAM modules. For instance, for a DDR3-800E module, the data bus is clocked at 400 MHz, i.e. a clock period of 2.5 ns, and hence $t_{RTRS} = \left\lceil \frac{4.5}{2.5} \right\rceil = 2$ cycles. For a DDR3-1600K, the data bus is clocked at 800 MHz, i.e. a clock period of 1.25, and hence $t_{RTRS} = \left\lceil \frac{4.5}{1.25} \right\rceil = 4$ cycles. Therefore, the controller proposed in this technical report reorders CAS commands in order to minimize the number of rank switches.

2.4 Related Work

A great deal of work has been done in the field of memories in real-time systems. For the sake of this technical report, however, we focus our discussion of the related work on real-time SDRAM

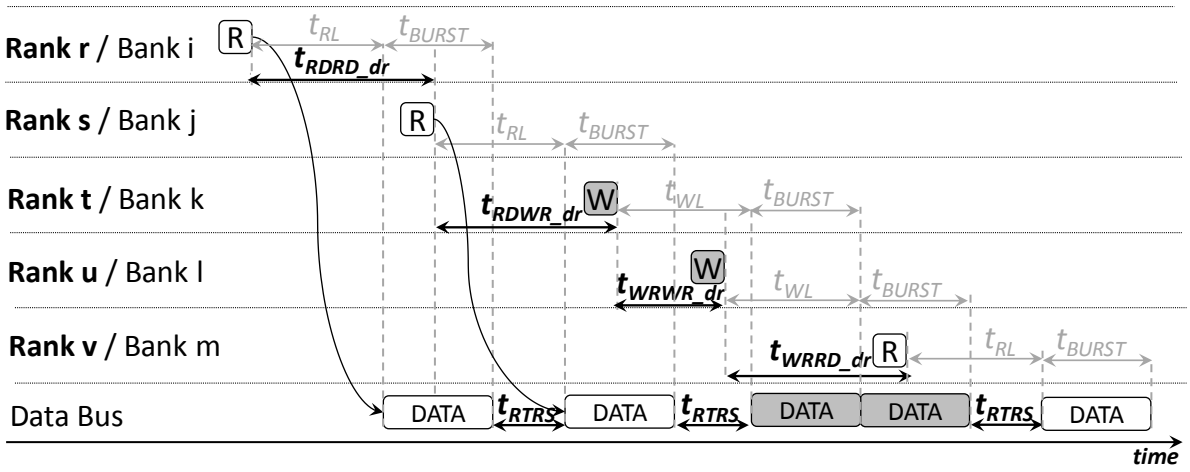


Figure 2.2: Graphical depiction of inter-rank timing constraints. Notice that two consecutive *write* transfers to different ranks do not demand a t_{RTRS} interval between them. This is because the same sender, i.e. the SDRAM controller, initiates both transfers.

controllers. As mentioned in the introduction, traditional real-time SDRAM controllers have employed a combination of the *close-row* policy and *pattern-based* bank interleaving [4, 5, 16]. In such controllers, each request is served with a statically precomputed command pattern that only exploits row buffer locality within the boundary of a single request, i.e. row buffers are precharged at the end of a request. Recent work in this area includes generation-independent scheduling [11] and supporting different request granularities [17, 18] (in addition to multiple granularities, [17] also considers mixed criticality).

However, for scenarios in which the the ratio between request granularity and data bus width is small (e.g. systems in which processors access a 64-bit SDRAM module to retrieve cache lines), researchers proposed *open-row* controllers. Such controllers do not precharge a row buffer at the end of a request and are frequently used in conjunction with a *private bank* setup for real-time tasks. The first work to openly discuss such strategy was [6, 7, 10] (the first reference discusses single-rank systems, the second multi-rank systems and the third provides an extended evaluation of the first). From now on, we will refer to it simply as the Waterloo controller, the name of the university in which the authors from [6, 7, 10] work. In summary, the Waterloo controller schedules CAS commands using two rules: firstly, inside a rank, the oldest CAS command has priority over other CAS commands, regardless of whether it causes a turnaround or not. And secondly, if there are multiple ranks, the CAS commands that have priority in their respective ranks compete with each other and are arbitrated using round-robin.

In single-rank systems, such rules can cause several data bus turnarounds. In multi-rank systems, they can cause a large number of rank switches. In high-speed SDRAM modules (which have larger timing constraints), such effects damage both worst-case and overall performance. Consequently, in [8, 9], we proposed an *open-row* controller that bundles *read* and *write* commands, hence minimizing the two aforementioned events.

It is important to notice that the timing analyses presented in [6, 7, 10] and in [8, 9] make one common assumption: the task under analysis has exclusive access to one or more SDRAM banks. This is not the case in the *open-row* controller from [19], which allows a hard real-time task to share a bank with several non-critical tasks and uses SPP (static priority preemptive) logic to benefit the former. Hence, for [19], guarantees for the hard real-time task assume that all requests miss at the row buffer. Also to be highlighted is the fact that, as a major revision of this technical report was being prepared, an article that proposes a mixed *row buffer* policy controller [20] was

accepted for publication: it uses a private bank mapping and *close-row* policy for hard real-time requestors and a shared bank mapping and *open-row* policy for soft real-time ones. The controller does not rely on *pattern-based* bank interleaving.

In comparison with the related work, our technical report is to our knowledge the first one to address generation-independent scheduling (thus including the DDR4 standard) and performance analysis from the perspective of *open-row* real-time SDRAM controllers.

3 Generic Notation for SDRAM Timing Constraints

In Sections 2.2 and 2.3, we discussed two types of timing constraints: *intra-device* (or *intra-rank*), which are a consequence of architectural features of SDRAM devices, and *inter-rank*, which are a consequence of two or more ranks sharing the data bus. These constraints were enumerated in Tables 2.1 and 2.2, respectively. As discussed in [11], the tables represent (in practical terms) a function that establishes the minimum distance between any two commands, taking into account whether such commands target the same rank, *bank group* and bank. We call such function d and formally define it below:

$$d(cmd_a, cmd_b, sameRank, sameGroup, sameBank) \tag{3.1}$$

where:

- cmd_a and $cmd_b \in$ {A, P, R, W}
- $sameRank,\ sameGroup,\ sameBank \in$ {Yes, No}

Taking into consideration that the DDR generation is implicitly given, we can now uniquely identify each timing constraint with an invocation of d. For instance, d(*W,R,No,No,No*) refers to the minimum distance between the execution of a *write* and the execution of a *read* in a different rank. Notice that such representation is well suited to describe algorithms, e.g. the ones used to statically generate the command *patterns* in [11]. However, if it were to be employed in equations in a timing analysis, the same representation would lack clarity.

Hence, we propose to represent any d-function invocation with the d prefix followed by a list of up to 5 arguments. The list of arguments employs the following syntax:

- The first two arguments are mandatory and define the pair of commands under consideration, which are represented by the letters A, P, R and W.
- The pair of commands is separated from the list of boolean arguments with a hyphen.
- Asserted boolean arguments are identified as: R (for *sameRank*), G (for *sameGroup*) and B (for *sameBank*). Notice that the hyphen in the previous item eliminates the ambiguity caused by the letter R representing both a *read* command and the *sameRank* argument.
- Non-asserted boolean arguments are identified as: $\overline{R}$, $\overline{G}$ and $\overline{B}$.
- *Don't care* values for boolean arguments are represented by omitting them.

For instance, $dPA\text{-}RGB$ represents the minimum distance between a *precharge* command followed by an *activate* command in the same rank, *bank group* and bank. Similarly, $dWR\text{-}\overline{R}$ represents the minimum distance between a *write* command followed by a *read* command in a different rank.

Notice that, unlike *sameRank* and *sameBank*, the *sameGroup* argument is not employed to actually index a cell table. Instead, it works as a cell value modifier for the DDR4 generation. For instance, $dAA\text{-}R\overline{G}B$ and $dAA\text{-}R\overline{GB}$ refer to the minimum distance between two consecutive *activate* commands to different banks in the same rank. If we look at Table 2.1, two commands fitting such description must be at least t_{RRD_x} cycles apart. If the *sameGroup* argument is true,

i.e. *dAA-RG*$\overline{B}$, we use the long version of the constraint (t_{RRD_L}). If the *sameGroup* argument is false, i.e. *dAA-R*$\overline{GB}$, we use the short version of the constraint (t_{RRD_S}). In systems that do not have the *bank groups* feature, e.g. DDR2 and DDR3, the group argument is simply ignored. As it will become clear, this powerful abstraction allows us to provide a single design and performance analysis for a SDRAM controller, independently of SDRAM generation.

Lastly, we make three important remarks about the proposed notation: firstly, notice that it lacks an expression to describe the distance between the execution of a *read* (or *write*) command and the start of the corresponding data transfer. For that purpose, we use *dRD* (*read* to data, which is depicted in Fig. 5.6) and *dWD* (*write* to data). In the DDR2/3/4 standards, the former refers to t_{RL} and the later to t_{WL}. Secondly, we use the expression *dCC* (potentially followed by a list of arguments) to refer to the minimum distance between any two consecutive CAS commands of the same type. For instance, *dCC-RG* refers to *dRR-RG* or *dWW-RG*. Finally, we point out that there is one constraint which cannot be represented using our notation: t_{FAW}, which represents a time window in which at most 4 *activate* commands can be executed within a rank. Hence, our timing analysis (Section 5.2) simply refers to it as t_{FAW}.

4 Multi-Generation SDRAM Scheduling

In this section, we propose a multi-generation *open-row* SDRAM controller architecture that supports arbitrary SDRAM module configurations, i.e. an arbitrary number of ranks (nR), number of *bank groups* per rank (nG)[1] and number of banks per rank (nB). Our architecture performs SDRAM command scheduling based on minimum distances between consecutive commands, which are discussed in Section 3. Moreover, it implements read/write bundling [8, 9] in order to minimize the number of data bus turnarounds and rank switching events.

4.1 SDRAM Controller Architectural Overview

We depict the architecture of our SDRAM controller in Fig. 4.1. The *bank groups* feature present in DDR4 is omitted for the sake of clarity, but is addressed in the text in our discussion about the channel scheduler (Section 4.3).

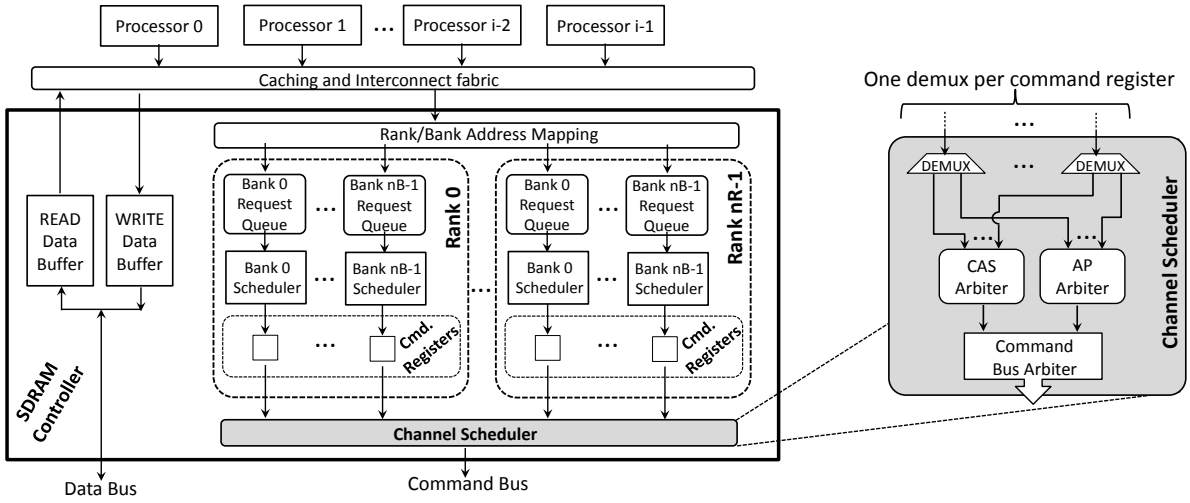


Figure 4.1: Logical architecture of our SDRAM controller. Notice that the controller has one bank request queue, one bank scheduler and one command register for each of the $nR \cdot nB$ banks of the controlled SDRAM module.

We now discuss the flow of requests through the controller. Firstly, incoming requests go through the address mapping block, which decodes their addresses and forwards them to the proper bank request queue. Requests are then removed one at a time from the queues by the corresponding bank scheduler, whose job is to generate the commands required to serve them. Such commands are then forwarded to the corresponding command register. The channel scheduler arbitrates between the command registers of all ranks and executes the selected command.

Before we proceed, we highlight that bank sharing (between two or more tasks) is out of the scope of this technical report. However, it could be accomplished by having two or more request queues for each bank, from which a round-robin arbiter would select the next request to be forwarded to the bank scheduler.

[1] To keep the architecture generic, we consider DDR2 and DDR3 memories to have a single group (nG=1) which comprises all banks of the system.

4.2 Bank Schedulers and Command Registers

The function of a bank scheduler is to translate a memory request into a set of SDRAM commands that fulfill such request. If the bank scheduler employs the *open-row* buffer policy, a request is translated into either a CAS command or into a *precharge-activate-CAS* sequence, depending on whether it hits or misses at the row buffer. The function of the command registers is to serve as an intermediate level of buffering that decouples the implementation of the channel scheduler from the bank schedulers. There is one command register for each bank scheduler. The channel scheduler removes commands from the registers when the commands are executed (sent to the SDRAM module). This allows the bank scheduler whose register was emptied to insert a new command (after the pertinent constraints no longer pose a violation), and so on.

A bank scheduler must only place a command in its register if such command can be immediately executed (in the cycle after the insertion) by the channel scheduler without violating any *exclusively intra-bank* timing constraints, i.e. timing constraints that rule the minimum distance between commands issued to the same bank and to the same bank only (those who have the RGB attribute). For instance, if the channel scheduler executes an *activate* from a command register, then the corresponding bank scheduler must wait at least $dAW\text{-}RGB - 1$ cycles before inserting a *write* into the aforementioned command register. The -1 term reflects the fact that if a command is inserted into a command register in instant t_0, even in the best case scenario such command will only be executed by the channel scheduler in instant $t_0 + 1$.

4.3 Channel Scheduler

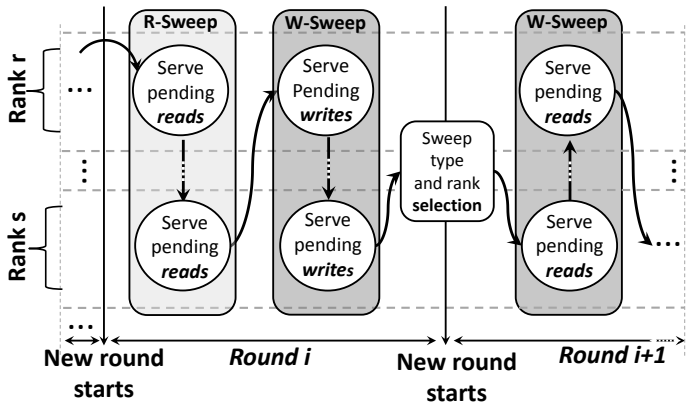
The channel scheduler has two functions: firstly, to arbitrate between and execute commands from the command registers, and secondly, to regularly refresh the SDRAM. In this technical report, we focus on the former and disregard the latter. We, however, refer the interested reader to [10] for a discussion on how to handle *refreshes* in *open-row* real-time controllers both from the architecture and timing analysis perspectives.

We depict a block diagram of the channel scheduler in the right portion of Figure 4.1. Notice that the arbitration of commands is performed in two layers: firstly, commands are arbitrated inside their corresponding arbiters (in the figure, notice the demultiplexers used to route a command to the proper arbiter). More specifically, *write* and *read* commands are routed to the CAS Arbiter, while *activate* and *precharge* commands are routed to the AP Arbiter. Then, commands that won the arbitration in their corresponding first layer arbiters are arbitrated by the Command Bus Arbiter. The remainder of this section discusses each of these arbiters individually. Before we proceed, however, we highlight again that the bank schedulers handle only *exclusively intra-bank* timing constraints. All the remaining constraints are handled by the channel scheduler.

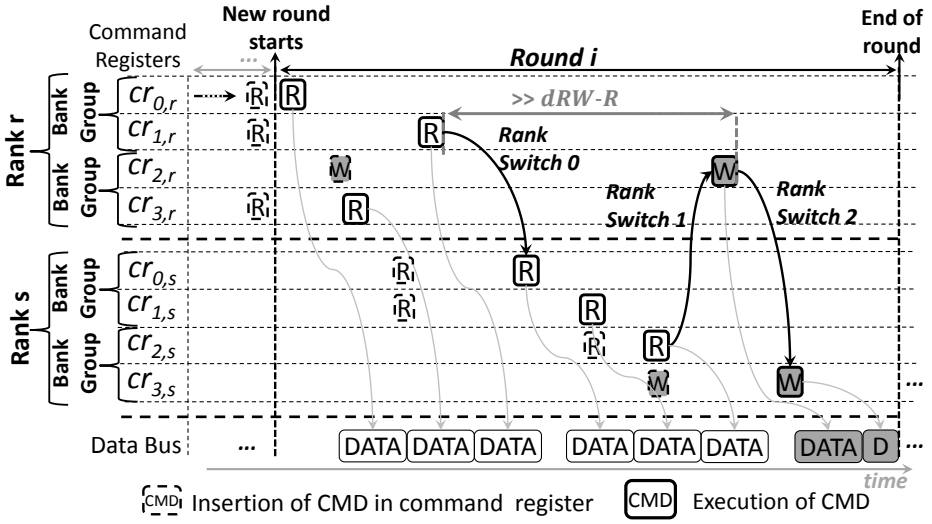
4.3.1 CAS Arbiter

The CAS Arbiter is where the *read/write* bundling is implemented. The arbiter operates with the concept of scheduling rounds, in which at most one pending CAS command (regardless whether a *read* or a *write*) from each command register is selected and forwarded to the Command Bus Arbiter. Each scheduling round is divided into a R-sweep and a W-sweep, as depicted in Fig. 4.2a. In a R-sweep, the arbiter serves at most one pending *read* for each command register of the rank currently being visited, a procedure to which we refer as visiting a rank. The R-sweep visits each rank at most one time and is over if all ranks have been visited (however, as we will later clarify, ranks that have no pending *read* commands can potentially suffer a so called *empty* visitation). The W-sweep performs the same operation, but for *write* commands.

We highlight three properties of the scheduler that are important for our timing analysis: (1) in the second sweep of each round (regardless whether *read* or *write*), the arbiter visits ranks in



(a) In terms of R- and W-Sweeps. Notice that each circle represents a rank visitation.



(b) In terms of commands. Notice that the insertion pattern of commands can prevent *group-interleaving*. This is the case in the scheduling of *read* commands in rank s because the *read* in cr_2, s arrives too late.

Figure 4.2: Example of read/write bundling in a hypothetical system with nR=2, nG=2 and nB=4.

the same order as it did in the first sweep. For instance, in round i in Fig. 4.2a, the R-Sweep visits ranks using $r \rightarrow ... \rightarrow s$ order. Hence, the W-Sweep from round i also visits ranks using $r \rightarrow ... \rightarrow s$ order. (2) In the first sweep of the round, the arbiter determines the first rank to be visited and the type of sweep operation by looking at the last rank that suffered a *non-empty* visitation in the previous round. In Fig. 4.2a, round $i+1$ starts with a W-Sweep firstly visiting rank s because round i finished with a W-Sweep in rank s. (3) If a rank has no pertinent valid commands at the time it is visited (e.g. a rank with no pending *writes* is visited during a W-Sweep), it suffers a a so-called *empty* visitation, i.e. it is marked as visited.

We now provide a pedagogical description on how command scheduling is actually performed inside a scheduling round (algorithms are presented in the end of this section). For that purpose, we provide an example in Fig. 4.2b. We make three observations about the figure. Firstly, for the sake of the example, each *bank group* contains only 2 banks (while in DDR4 systems each *bank group* contains 4 banks). Secondly, whenever possible, rank switches are scheduled concurrently with bus turnarounds. For instance, rank switch 0 allows rank s to be visited while rank r is constrained due to dRW-R. And thirdly, whenever possible, our scheduler performs *group*

interleaving, i.e. it prevents the execution of two consecutive CAS commands that target the same *bank group*. This is the case, for instance, in the scheduling of *read* commands in rank r. However, depending on the insertion pattern of commands, a *group interleaved* execution pattern is not possible. This is the case, for instance, in the scheduling of *read* commands in rank s. More specifically, the *read* command in $cr_{2,s}$ is only inserted after the *reads* from $cr_{0,s}$ and $cr_{1,s}$ are executed. This observation is very important because, in order to extract worst-case bounds, we need to take into account that the insertion pattern of CAS commands limits the possibility of *group interleaving*.

We now discuss the logical architecture of the CAS Arbiter. In order to implement read/write bundling, the CAS Arbiter requires a state. As depicted in Fig. 4.3, the state is comprised of the *served flags* vector, which indicates whether a command register was already served in the current round, the *current rank* register, which indicates the current rank being visited, and the *bundling type* register, which indicates the type of the sweep operation currently ongoing (R or W). The *served flags* vector enforces that each command register is selected at most once every round. The *current rank* and the *bundling type* registers control the sweeping operations.

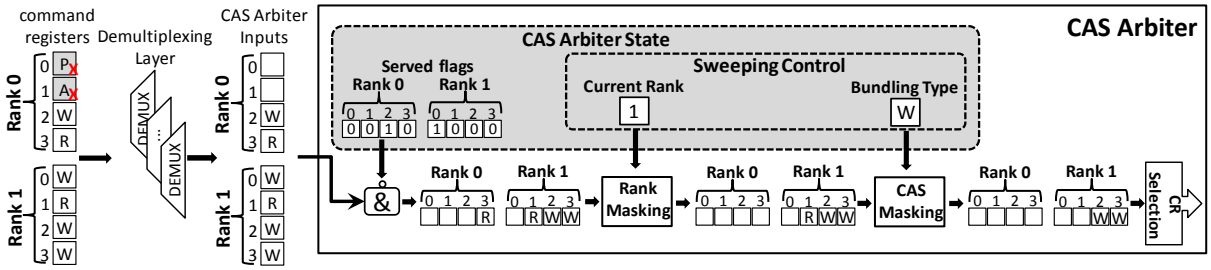


Figure 4.3: Example of operation of the CAS Arbiter for a system with nR=2 and nB=4. Notice that only CAS commands arrive at the input of the arbiter (*activates* and *precharges* are routed to the AP Arbiter). Moreover, notice that for the sake of simplicity, the *bank groups* feature and the logic that updates the state of the arbiter are omitted.

We enumerate the steps performed by the CAS Arbiter. Firstly, the CAS Arbiter masks out the command registers that have already been served in the round. This is performed using the *served flags* vector. In the second and third steps, the arbiter performs rank and CAS masking, which masks out commands from ranks that do not match the *current rank* and *bundling type* registers. In the figure, *current rank* is 1 and *bundling type* is W and, consequently, only *write* commands from rank 1 are left unmasked. In the last step, (the CR Selection), the arbiter prioritizes the oldest CAS command that can be immediately executed without violating a timing constraint.

We describe how the state is updated. We firstly discuss the *served flags* vector. Every time a CAS is selected and executed by the Command Bus Arbiter, the corresponding bit in the *served flags* vector is set. Furthermore, the vector is cleared every time a new round starts. A new round starts after the scheduler completes both a R- and a W-sweep. The *bundling type* register is always flipped (from W to R, or vice versa) when a sweep was just completed, but the round is not over (in Fig. 4.2a, this is the case after the visitation of rank s in the R-Sweep from round i). The *current rank* is updated when the output of the CAS Masking stage is null using the properties discussed in the beginning of this subsection.

In order to avoid missunderstandings, we now complement our pedagogical description of the CAS Arbiter with Agorithm 1, which formalizes the operation of the CAS Arbiter using pseudo-code. We make the following observations about the pseudo-code: firstly, comments are depicted with grey font after // symbols. Moreover, our pseudo-code is heavily commented for ease of understanding. Secondly, our pseudo-code uses a objected-oriented notation. For instance, *cmd.rank*

and *cmd.bank* are used respectively to refer to the target bank and the target rank of a command. Thirdly, code that initializes the control registers has been left out. Fourthly, the difference between a procedure and a function is that the latter returns a value (to be employed by the function caller), while the former returns nothing. And finally, the Operate( ) procedure describes the operation of the CAS Arbiter (which consists of performing scheduling rounds).

Algorithm 1 CAS Arbiter Operation

```
1:  CAS Arbiter Control Registers
2:      // parameters of SDRAM module configuration
3:      nR, nG, nB;
4:      // For sweeping and rank visitation control
5:      served_flags[nR][nB], current_rank, bundling_type;
6:      last_cmd[nR]; // Holds the last executed CAS command (per rank)
7:      last_exec_cmd; // Last executed CAS command (regardless of the rank)
8:  end CAS Arbiter Control Registers
9:
10: List of Command Registers
11:     crs[nR][nB];
12: end List of Command Registers
13:
14: // Basic operation of CAS Arbiter
15: procedure Operate( )
16:     while True do
17:         Perform_Scheduling_Round( )
18:
19: procedure Perform_Scheduling_Round( )
20:     // Resets (clears) the served_flags vector
21:     for i ← 0; i < nR; i ← i + 1 do;
22:         for j ← 0; j < nB; j ← j + 1 do;
23:             server_flags[i][j] ← False;
24:
25:     // The last command executed in the previous round defines the type of the first
26:     // sweep operation and the first rank to be visited.
27:     current_rank ← last_exec_cmd.rank // last rank to suffer non-empty visitation
28:     bundling_type ← last_exec_cmd.type // same type as last command
29:     first_visited_rank ← current_rank
30:
31:     // First Sweep operation of the round
32:     // Example: in a system with nR=4, if the first rank to be visited is 2, the arbiter
33:     // visits ranks using the following order: 2 → 3 → 0 → 1
34:     do
35:         Visit_Rank( ) // Visits the current rank
36:         // Continues in the next page...
```

```
37:         current_rank ← (current_rank + 1) mod nR
38:     while current_rank ≠ first_visited_rank
39:
40:     // (Implementation alternative for the first sweep: as long as the first rank to be visited
        matches the last rank to suffer a non-empty visitation in the previous round, a different visitation
        order could be exploited in the first sweep.)
41:
42:     // After the first sweep, we flip the bundling_type register ...
43:     if bundling_type = Read then
44:         bundling_type ← Write
45:     else
46:         bundling_type ← Read
47:
48:     // Second Sweep operation of the round (at this point, current_rank is equal to
49:     // first_visited_rank). Notice that if the first sweep visited ranks using 2 → 3 → 0 → 1
50:     // order, so should the second sweep.
51:     do
52:         Visit_Rank( ) // Visits the current rank
53:         current_rank ← (current_rank + 1) mod nR
54:     while current_rank ≠ first_visited_rank
55:     return
56:
57: // The rank to be visited and the sweep type are determined by the current_rank
58: // and bundling_type registers.
59: // Notice that empty visitations happen when None is returned
60: // the first time Find_Next_CAS( ) is called.
61: procedure Visit_Rank( )
62:     winningcmd ← Find_Next_CAS( )
63:
64:     while winningcmd ≠ None do
65:         while (winningcmd cannot be executed without violating a constraint) do
66:             wait until next cycle
67:         Execute_Command(winningcmd)
68:         winningcmd ← Find_Next_CAS( )
69:
70:     // If we reach this point, then the visitation is over
71:     return
72:
73: // Finds the next CAS that matches bundling_type and current_rank
74: // (Returns None if a rank visitation is over)
75: function Find_Next_CAS( )
76:     // This variable will hold the oldest pending command in the same bank group (sbg)
77:     // as the last executed CAS command in current_rank
78:     oldest_cmd_sbg ← None
79:     // Continues in the next page...
```

```
80:     // This variable will hold the oldest pending command in a different bank group (dbg)
81:     // than the last executed CAS command in current_rank
82:     oldest_cmd_dbg ← None
83:
84:     // Now we fill the oldest_cmd_sbg and oldest_cmd_dbg variables
85:     // For that purpose, we iterate over the cmd. registers from the rank being visited
86:     for i ← 0; i < nB; i ← i + 1 do;
87:         cmd ←crs[current_rank][i].cmd // Gets the command from the command register
88:         if cmd ≠ None then
89:             if cmd.type = bundling_type and served_flags[current_rank][i] = False then
90:                 if bank i in same group as bank from last_cmd[current_rank] then
91:                     oldest_cmd_sbg ← Select_Oldest(oldest_cmd_sbg, cmd)
92:                 else
93:                     oldest_cmd_dbg ← Select_Oldest(oldest_cmd_dbg, cmd)
94:
95:     // Command that wins arbitration (we prioritize the cmd for a different bank group)
96:     if oldest_cmd_dbg ≠ None then
97:         winningcmd ← oldest_cmd_dbg
98:     else
99:         winningcmd ← oldest_cmd_sbg
100:     return winningcmd
101:
102: procedure Execute_Command(cmd)
103:     Send_Command_To_Command_Bus_Arbiter(cmd)
104:
105:     // As CAS commands have priority over activates and precharges we know they are
106:     // immediately executed so we can already update the control registers and return...
107:     served_flags[cmd.rank][cmd.bank] ← True
108:     last_cmd[cmd.rank] ← cmd
109:     last_exec_cmd ← cmd
110:
111:     return
112:
113: // If cmda = None and cmdb = None, this function also returns None.
114: function Select_Oldest(cmda,cmdb)
115:     if cmda = None then
116:         retcmd ← cmdb
117:     else if cmdb = None then
118:         retcmd ← cmda
119:     else
120:         // Compares the insertion timestamps (in the corresponding command registers)
121:         if cmda.insertiontimestamp < cmdb.insertiontimestamp then
122:             retcmd ← cmda
123:         else
124:             retcmd ← cmdb
125:     return retcmd
```

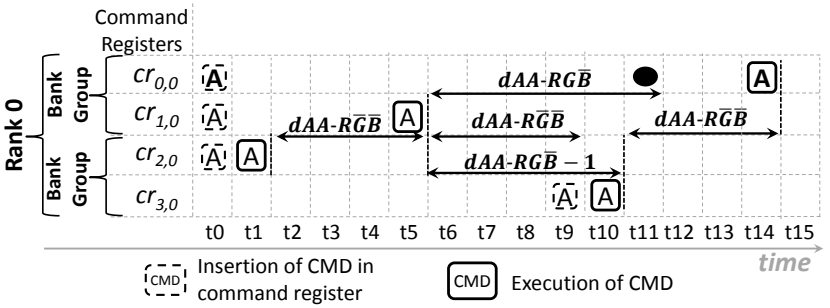
4.3.2 AP Arbiter

Although not depicted in Fig. 4.1, the AP Arbiter is logically divided into two arbiters: the P Arbiter (for *precharges*) and the A Arbiter (for *activates*). Both the P Arbiter and the A Arbiter only output commands that can be immediately executed without violating timing constraints. Consequently, the AP Arbiter simply prioritizes the oldest command in order to select between the output of the P Arbiter and the A Arbiter.

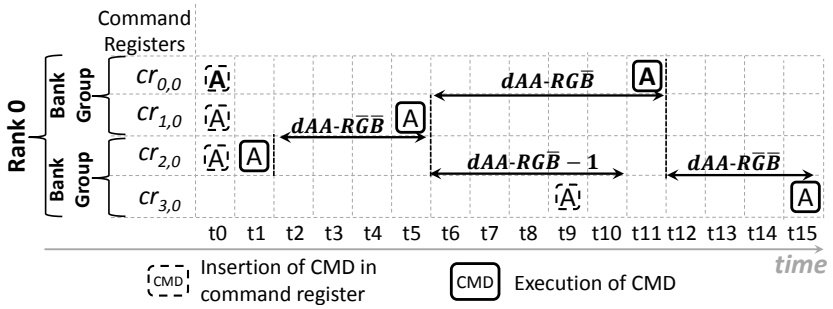
We now discuss the P Arbiter and the A Arbiter individually. The P Arbiter simply prioritizes the oldest pending *precharge*. The A Arbiter is more complex: at the level of a rank, it employs what we call *real-time aware oldest ready* arbitration, and at the level of the module, it selects the oldest *activate* which won the arbitration in its corresponding rank.

We now discuss the intra-rank arbitration of the A Arbiter. Inside a rank, the A Arbiter must take into consideration $dAA\text{-}R\overline{B}$ (with the G or $\overline{G}$ arguments for DDR4) and the t_{FAW} constraint (see Section 3). As t_{FAW} requires knowing the history of previous *activate* commands, the A Arbiter makes scheduling decisions based on $dAA\text{-}R\overline{B}$, and then simply holds the winning command until t_{FAW} no longer poses a violation. Consequently, the remaining of our discussion will focus on $dAA\text{-}R\overline{B}$ (we will, however, account for t_{FAW} in the timing analysis).

As previously mentioned, the intra-rank portion of the A Arbiter uses *real-time aware oldest ready* arbitration. The *oldest ready* expression means the oldest *activate* that can be immediately executed has priority. The *real-time aware* expression means that the *ready* requirement (being able to be immediately executed) is ignored in one specific situation: if more than $dAA\text{-}R\overline{GB}$ cycles have passed since the last execution of an *activate* command. Such exception is to prevent the situation depicted in Fig. 4.4a.



(a) Oldest Ready arbitration.



(b) Real-time aware oldest ready arbitration.

Figure 4.4: Intra-rank arbitration of *activates* in a scenario in which $dAA\text{-}RG\overline{B} = 6$ and $dAA\text{-}R\overline{GB} = 4$. The effect of t_{FAW} is deliberately ignored, but is accounted for in the timing analysis. Moreover, in (a), the solid black circle represents the moment in time in which the *activate* from $cr_{0,0}$ would be executed if the *ready* requirement was ignored.

In the figure, the *activate* in $cr_{0,0}$ is blocked by three interfering *activates*. However, because

the *activate* in $cr_{3,0}$ arrives late (in $t9$), the time interval between its execution with regard to the *activate* from $cr_{1,0}$ is larger than $dAA\text{-}R\overline{GB}$. From the worst-case latency perspective, this would mean that, inside its own rank, an *activate* command could be blocked by $nB-1$ interfering *activates* and that each of this blockings would amount to $dAA\text{-}RG\overline{B}-1$.

Hence, we employ a *real-time aware oldest ready* arbitration, which is depicted in Fig. 4.4b. In the figure, notice that the *activate* in $cr_{0,0}$ is only blocked by $nB-2$ interfering *activates*. More importantly, the only way for it to be blocked $nB-1$ times would be if all interfering *activates* were previously available, in which case a blocking of $dAA\text{-}R\overline{GB}\cdot(nB-1)$ cycles would be observed. Notice also that, in systems with $nG=1$, *oldest ready* and *real-time aware oldest ready* produce the same outcome.

Finally, in order to avoid missunderstandings, we formalize the operation of the A Arbiter using pseudo-code in Algorithm 2. Moreover, we make two final observations: firstly, we do not provide algorithmic descriptions of the P Arbiter and of the AP Arbiter, as they are trivial. And secondly, from the perspective of the pseudo-code employed in Algorithm 2, the same observations as for Algorithm 1 (see Section 4.3.1) apply.

Algorithm 2 Operation of the ACT Arbiter

```
1:  A Arbiter Control Registers
2:      // To implement group-interleaving
3:      last_cmd[nR]; // Holds the last executed activate (per rank)
4:      last_exec_cmd; // Last executed activate (regardless of the rank)
5:      current_clock_tick; // Keeps track of time
6:  end A Arbiter Control Registers
7:
8:  List of Command Registers
9:      crs[nR][nB];
10: end List of Command Registers
11:
12:
13: // Describes basic operation of the A Arbiter
14: procedure Operate( )
15:     while True do
16:         cmd ← Find_Next_ACT_to_Be_Executed( )
17:         if cmd ≠ None then
18:             Send_ACT_To_AP_Arbiter(cmd)
19:             // In the AP Arbiter, cmd can be blocked by other precharges or CAS commands.
20:             // Hence, we wait....
21:             wait until command is executed
22:             // Now we update the control registers...
23:             last_cmd[cmd.rank] ← cmd
24:             last_exec_cmd ← cmd
25:
26:
27: // Continues in next page...
```

4.3.3 Command Bus Arbiter

The command bus can only carry one command per cycle and, hence, needs to be arbitrated. As the CAS Arbiter and the AP Arbiter only forward commands that can be immediately executed,

```
28: // Searches all ranks and returns an activate command that can
29: // be immediately executed without violating a constraint
30: // (or None, if no activate can be immediately executed)
31: function Find_Next_ACT_to_Be_Executed( )
32:     // Firstly we compile a list of pending activates (one per rank)
33:     activate_commands[nR] ← new Activate_Vector[nR];
34:     for i ← 0; i < nR; i ← i + 1 do;
35:         activate_commands[i] ←Find_Next_ACT_in_Rank(i)
36:
37:     // Searches all ranks and selects oldest pending activate.
38:     winningcmd ← None
39:     for i ← 0; i < nR; i ← i + 1 do;
40:         if activate_commands[i] can be immediately executed without violating a
  constraint then
41:             winningcmd ← Select_Oldest(activate_commands[i],winningcmd)
42:
43:     return winningcmd
44:
45: // Selects an activate command from an specific rank using
46: // real-time aware oldest ready arbitration
47: function Find_Next_ACT_in_Rank(rank_id)
48:     // This variable will hold the oldest pending activate in the same bank group (sbg)
49:     // as the last activate executed in rank_id
50:     oldest_cmd_sbg ← None
51:     // This variable will hold the oldest pending activate in a different bank group (dbg)
52:     // as the last activate executed in rank_id
53:     oldest_cmd_dbg ← None
54:
55:     // Now we fill the oldest_cmd_sbg and oldest_cmd_dbg variables
56:     // For that purpose, we iterate over the cmd. registers from the rank under consideration
57:     for i ← 0; i < nB; i ← i + 1 do;
58:         cmd ←crs[rank_id][i].cmd // Gets the command from the command register
59:         if cmd ≠ None then
60:             if cmd.type = Activate then
61:                 if bank i in same group as bank from last_cmd[rank_id] then
62:                     oldest_cmd_sbg ← Select_Oldest(oldest_cmd_sbg, cmd)
63:                 else
64:                     oldest_cmd_dbg ← Select_Oldest(oldest_cmd_dbg, cmd)
65:
66:
67:     // Continues in next page...
```

```
68:     if oldest_cmd_dbg = None then
69:         winningcmd ← oldest_cmd_sbg
70:     else if oldest_cmd_sbg = None then
71:         winningcmd ← oldest_cmd_dbg
72:     else
73:         // At this point, we know oldest_cmd_sbg and oldest_cmd_dbg
74:         //are different than None
75:         // The real-time aware oldest ready arbitration is performed here
76:         if  (oldest_cmd_sbg is older than oldest_cmd_dbg and
77:           (current_clock_tick − last_exec_cmd.execution_timestamp) > dAA-RGB) then
78:             winningcmd ← oldest_cmd_sbg
79:         else
80:             winningcmd ← oldest_cmd_dbg
81:     return winningcmd
82:
83: // If cmda = None and cmdb = None, this function also returns None.
84: function Select_Oldest(cmda,cmdb)
85:     if cmda = None then
86:         retcmd ← cmdb
87:     else if cmdb = None then
88:         retcmd ← cmda
89:     else
90:         // Compares the insertion timestamps (in the corresponding command registers)
91:         if cmda.insertiontimestamp < cmdb.insertiontimestamp then
92:             retcmd ← cmda
93:         else
94:             retcmd ← cmdb
95:     return retcmd
```

the Command Bus Arbiter employs a simple fixed priority scheme: commands from the CAS Arbiter have priority over the ones from the AP Arbiter. This ensures that *reads* and *writes* do not suffer any interference.

5 Timing Analysis

In this section, we compute the worst-case cumulative SDRAM latency of a task (L_{Task}^{SDRAM}), i.e. the the maximum amount of time that a task spends idle while waiting for its memory requests to be served. Our timing analysis is generic and works regardless of the DDR generation and configuration of the SDRAM module (nR, nG and nB). For the sake of notation, we consider SDRAM generations that do not have the *bank groups* feature, e.g. DDR2 and DDR3, to have nG=1.

This section is structured as follows: firstly, in Section 5.1, we present our assumptions. Then, in Section 5.2, we compute the worst-case latency of individual SDRAM commands. Finally, in Section 5.3, we use the computed command latencies to calculate the worst-case latency of requests, which are then combined to compute the worst-case SDRAM latency of a task.

5.1 Assumptions

Our timing analysis relies on the following assumptions: 1) the processor running the task under analysis (u.a.) relies on caches and only accesses the SDRAM to retrieve or forward cache lines. 2) The processor is fully timing compositional [21], which means that it uses in-order execution and stalls at every memory request, including writes. This enforces that a request only arrives at the SDRAM controller after the previous request from the same task has been served. 3) No cache related effects change the number of cache misses experienced by the task u.a.. Hence, if multi-tasking is employed, we assume that the task u.a. is assigned an exclusive partition in the cache. 4) No multi-tasking related effects cause destruction of row buffer locality at the bank being used by the task u.a.. Hence, if interfering tasks are co-executed with the task u.a. (e.g. in different processing cores), they will not compete for the same bank in the SDRAM.

Finally, we highlight that we assume no knowledge about interfering tasks on the system. This is a desirable feature, because the computed bound remains valid regardless of activity of interfering requestors.

5.2 Worst-case Latency of Commands

The worst-case latency of a command, to which we refer as L^{CMD}, refers to the largest observable timing interval between the insertion of a command into a command register and its execution by the channel scheduler. In this subsection, we calculate the worst-case latencies of *read*, *write*, *activate* and *precharge* commands, to which we respectively refer as L^R, L^W, L^A and L^P.

Please notice that the equations in this subsection employ the following notation: $\max\begin{Bmatrix}A\\B\end{Bmatrix}$ returns the largest value between A and B, which is also represented with $\max\{A, B\}$. Moreover, $\begin{cases}\textbf{if } cond \textbf{ then: } A\\ \textbf{else then: } B\end{cases}$ returns A if *cond* is true, and B otherwise.

We now discuss the worst-case latency of a *read* command. For that purpose, we firstly state and proof Lemma 1.

Lemma 1. *Given a sequence of $n \leq nB$ CAS commands of the same type that are consecutively executed in different banks of the same rank, the maximum timing interval between the execution of the first and the last command of such sequence is given by Eq. 5.1.*

$$CC_{sum}(n) = \begin{cases} \textbf{\textit{if}} \;\; nG = 1 \;\textbf{\textit{then:}} & (n-1)\cdot dCC\text{-}R \quad \textit{if} \\ \textbf{\textit{else then:}} & (n-1)\cdot dCC\text{-}RG + \left\lfloor \frac{(n-1)}{\left(\frac{nB}{nG}\right)} \right\rfloor \cdot (dCC\text{-}R\overline{G} - dCC\text{-}RG) \end{cases} \tag{5.1}$$

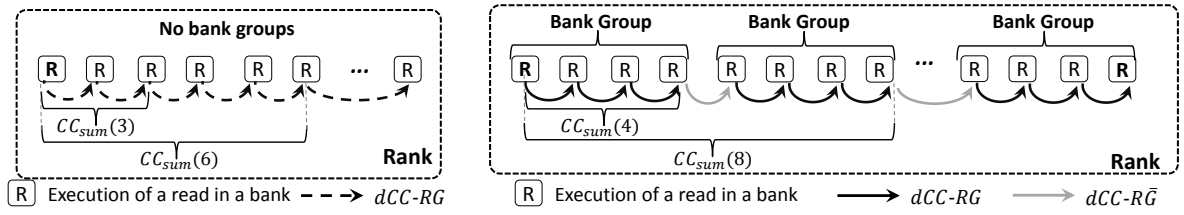
Proof. We prove the lemma by construction using Figs. 5.1a and 5.1b, which refer respectively to the case in which $nG = 1$ and the case in which $nG \geq 2$, respectively. In the figures, latencies are represented using directed edges that connect two commands executed consecutively. Examples of the latencies accounted by CC_{sum} for arbitrary inputs are depicted at the bottom of the figures.

The computation of CC_{sum} in systems with $nG = 1$ is simple, as only one type of edges must be accounted (see Fig. 5.1a). In such case, the number of accounted $dCC\text{-}R$ edges is equal to the number of commands in the sequence subtracted by one. For instance, $CC_{sum}(6) = 5 \cdot dCC\text{-}R$.

The computation of CC_{sum} in systems with $nG \geq 2$ is slightly more complex, as it demands taking into account *group interleaving*. More specifically, consecutive commands executed in the same *bank group* take longer to execute ($dCC\text{-}RG \geq dCC\text{-}R\overline{G}$). Hence, in order to compute a safe bound, we need to assume that the insertion pattern of interfering CAS commands minimizes the possibility of performing *group interleaving* (see Section 4.3.1). Thinking in graphical terms (see Fig. 5.1b), CC_{sum} must reflect scenarios in which number of solid black edges ($dCC\text{-}RG$) is maximized and the number of solid gray arrows is minimized. Notice that the pattern assumed in Fig. 5.1b clearly fulfills such goal, as more solid black edges would only be possible if more than one CAS command could be executed in the same bank (which is not addressed by the lemma).

Hence, we now focus on proving that the computation of CC_{sum} in systems with $nG \geq 2$ accurately describes such pattern. For such purpose, notice that the expression that computes CC_{sum} in systems with $nG \geq 2$ has two terms: the first one assumes that the latency between any two consecutive CAS commands is $dCC\text{-}RG$ (solid black edges), which is overly conservative. The second term corrects such overly conservative assumption.

More specifically, if n is larger than the number of banks per *bank group* (which is given by nB/nG), at least one pair of consecutive commands will cross the *bank group* boundary. The second term of the equation simply identifies how many command pairs fall into such category and then replaces occurrences of $dCC\text{-}RG$ by $dCC\text{-}R\overline{G}$ accordingly. The -1 in the upper part of the fraction inside the *floor* function is necessary because $dCC\text{-}R\overline{G}$ only occurs if n is larger (and not larger or equal) than the number of banks per *bank group*. This concludes our proof. □



(a) In systems with $nG = 1$.

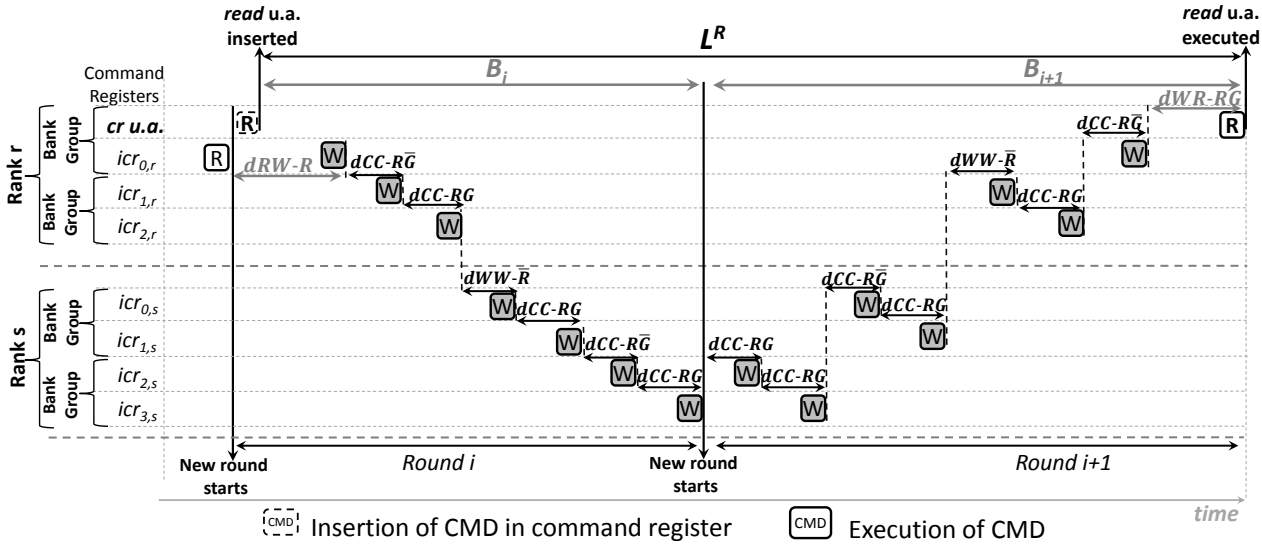
(b) In systems with $nG \geq 2$.

Figure 5.1: Graphical depiction of the CC_{sum} function. Notice that $CC_{sum}(n) \neq 2 \cdot CC_{sum}(n/2)$. Such property will become important to understand the worst-case latency of CAS commands.

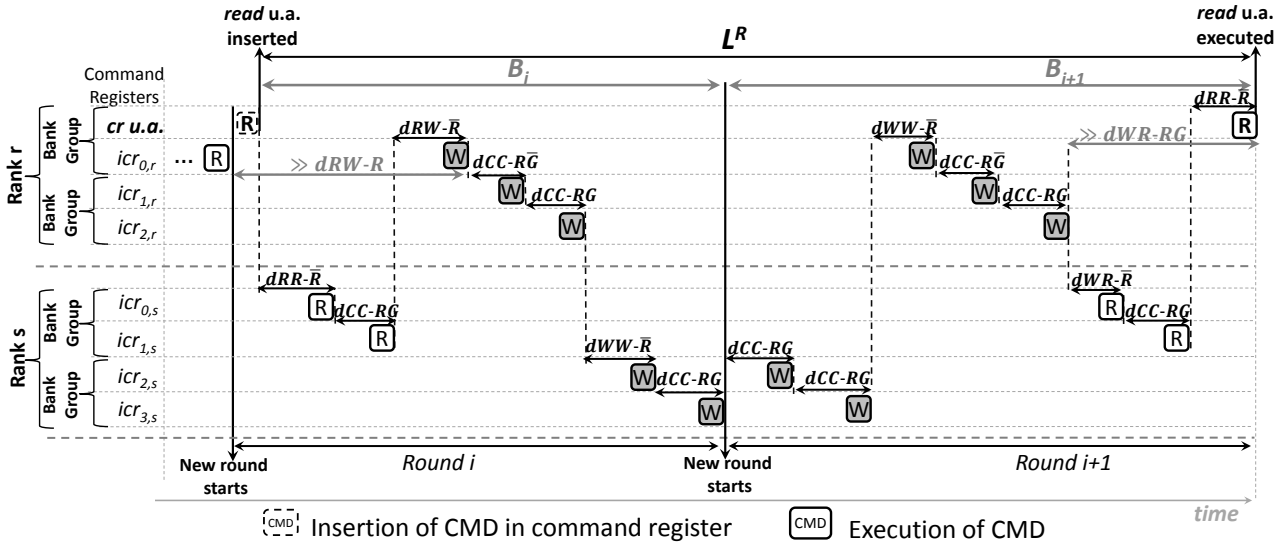
We now discuss L^R. In order to compute the worst-case latency of a *read* command, we must assume that such command is blocked twice by each interfering command register. Such scenario is possible if the *read* u.a. arrives exactly after the decision to end a rank visitation is made (here we correct a non-conservative assumption of the *read* u.a. arriving as close as possible to the execution of the previous CAS command in the bank u.a. made in [8, 9]).

Moreover, in order to reach a safe bound, there are two cases that we must consider: one in which data bus turnarounds are fully experienced and another one in which the data bus turnarounds run concurrently with rank switches. We refer to the former as *case A* and depict it for a dual-rank system in Fig. 5.2a. We refer to the latter as *case B* and depict it for a dual-rank system in

Fig. 5.2b. Our analysis must account for both. That being said, we state Theorem 1.



(a) Case A.



(b) Case B.

Figure 5.2: Cases that must be considered in order to compute the worst-case latency of a *read* command in a hypothetical system with nR=2, nG=2 and nB=4. In order to keep the figures clear, only the insertion of the command u.a. is depicted (for the interfering command registers, we only depict the execution).

Theorem 1. *The worst-case latency of a* read *command is calculated with Eq. 5.2.*

$$L^R = B^i + B^{i+1} \tag{5.2}$$

where:

$$B^i = \max \begin{cases} ccds_i(CaseA) + switches_i(CaseA) \\ ccds_i(CaseB) + switches_i(CaseB) \end{cases} \tag{5.3}$$

$$B^{i+1} = \max \begin{cases} ccds_{i+1}(CaseA) + switches_{i+1}(CaseA) \\ ccds_{i+1}(CaseB) + switches_{i+1}(CaseB) \end{cases} \tag{5.4}$$

$$switches_i(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \;\; dRW\text{-}R + dWW\text{-}\overline{R} \cdot (nR-1) \\ \textbf{\textit{else then:}} \;\; \max \begin{cases} \max\{(dRR\text{-}\overline{R} \cdot (nR-1) + dRW\text{-}\overline{R}), dRW\text{-}R\} \;\; + dWW\text{-}\overline{R} \cdot (nR-1) \\ dRR\text{-}\overline{R} \cdot (nR-1) + \max\{(dRW\text{-}\overline{R} + dWW\text{-}\overline{R} \cdot (nR-1)), dRW\text{-}R\} \end{cases} \end{cases} \tag{5.5}$$

$$ccds_i(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \; CC_{sum}(nB-1) + CC_{sum}(nB) \cdot (nR-1) \\ \textbf{\textit{else then:}} \;\; CC_{sum}(nB-1) + 2 \cdot CC_{sum}(nB/2) \cdot (nR-1) \end{cases} \tag{5.6}$$

$$switches_{i+1}(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \;\; dWW\text{-}\overline{R} \cdot (nR-1) + dWR\text{-}RG \\ \textbf{\textit{else then:}} \;\; \max \begin{cases} \max\{(dWW\text{-}\overline{R} \cdot (nR-1) + dWR\text{-}\overline{R}), dWR\text{-}RG\} \;\; + dRR\text{-}\overline{R} \cdot (nR-1) \\ dWW\text{-}\overline{R} \cdot (nR-1) + \max\{(dWR\text{-}\overline{R} + dRR\text{-}\overline{R} \cdot (nR-1)), dWR\text{-}RG\} \end{cases} \end{cases} \tag{5.7}$$

$$ccds_{i+1}(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \;\; CC_{sum}(nB-1) + dCC\text{-}RG + CC_{sum}(nB-1) \cdot (nR-1) \\ \textbf{\textit{else then:}} \;\; CC_{sum}(nB-1) + dCC\text{-}RG + 2 \cdot CC_{sum}(nB/2) \cdot (nR-1) \end{cases} \tag{5.8}$$

Proof. The latency of the *read* u.a. is a consequence of the blocking experienced by it in two consecutive rounds of the CAS Arbiter: round i, in which the *read* u.a. arrives (late enough not to be served), and round $i+1$, in which the *read* u.a. is executed. We refer to the blocking experienced in such scheduling rounds as B_i and B_{i+1}, respectively.

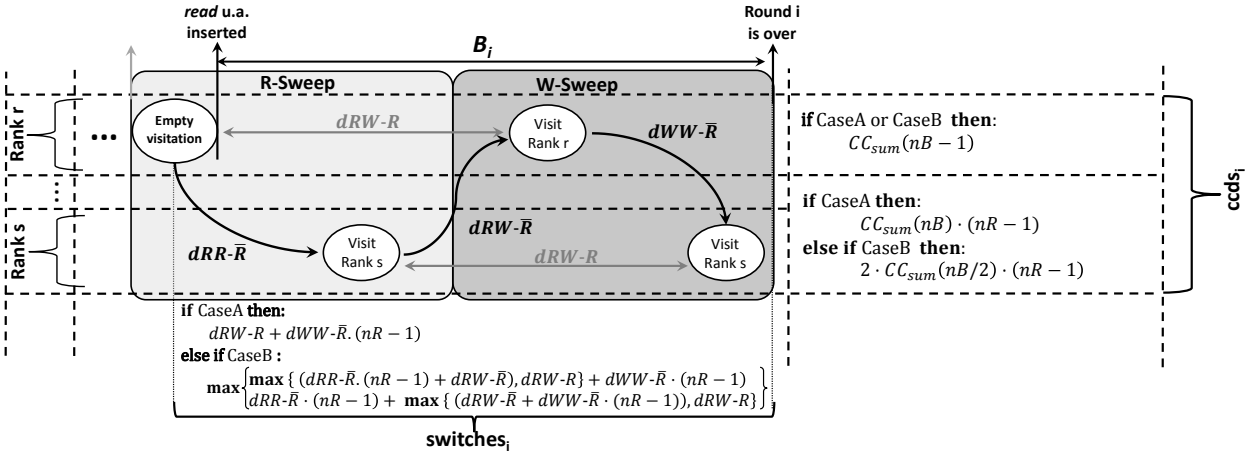
In order to compute B_i and B_{i+1}, it is useful to observe that they both obey patterns. We depict such patterns in Figs. 5.3a and 5.3b, respectively, and make three important considerations about them: firstly, the scenario depicted in Fig. 5.2a does follow the pattern depicted in Figs. 5.3a and 5.3b (although at first glance it might not look like it, as rank s suffers *empty* visitations during R-Sweeps). Secondly, Figs. 5.3a and 5.3b show that round i ends with a W-Sweep in rank s and that round $i+1$ starts with a W-Sweep in rank s. If rank s had no pending *write* command at the beginning of round $i+1$, such rank would only be visited again in a W-Sweep in round $i+2$.

And finally, each pattern divides the computation of blocking into two components: a *ccds* component and a *switches* component. The former accounts for dCC latencies and is depicted at the right of the corresponding figures (Figs. 5.3a and 5.3b). The latter accounts for rank switching latencies (which overlap with data bus turnarounds) and is depicted at the bottom of the corresponding figures. In order to identify each component uniquely, we use subscripts, e.g. $ccds_i$ refers to the *ccds* component of B_i.

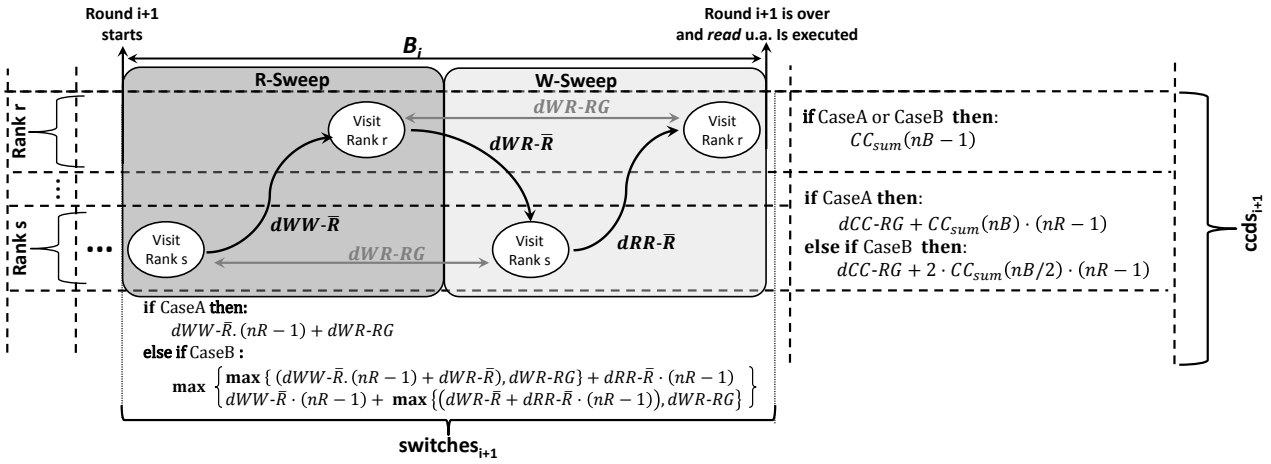
We now discuss the equations enunciated by our theorem. Eqs. 5.10 and 5.11 compute B_i and B_{i+1} by selecting the combination of *ccds* and *switches* that leads to the largest latency, i.e. either the one depicted in Fig. 5.2a (case A) or the one depicted in Fig. 5.2b (case B). As they are simple, we provide no further discussion about them.

Eqs. 5.14, 5.15, 5.12 and 5.13 are more sophisticated: they compute the *ccds* and *switches* components as a function of the case under consideration (case A or case B). We firstly discuss the correctness of Eq. 5.14, which computes the *switches* component of round i. If we consider case A (e.g. Fig. 5.2a), then a full data bus turnaround from read to write is experienced ($dRW\text{-}R$). Moreover, each interfering rank is visited in a W-Sweep and, hence, we account for the $(nR-1)$ occurrences of $dWW\text{-}\overline{R}$.

If, however, we consider case B (e.g. Fig. 5.2b), the computation is more complex because the turnaround from *read* to *write* runs simultaneously with the rank switches. In Fig. 5.3a, notice



(a) Blocking experienced in round i.



(b) Blocking experienced in round $i+1$.

Figure 5.3: Pattern observed for the worst-case blocking of a *read* command. Notice that the computation of $ccds_i$ and $ccds_{i+1}$ (right portion of the figures) is divided into two parts: the upper one, which accounts for dCC latencies in the rank u.a. (i.e. the rank containing the cr u.a.), and the lower one, which accounts for dCC latencies in interfering ranks.

that the turnaround can take place in any rank of the system (the figure depicts it in rank r and in rank s). In order to be conservative, we have to find the combination that leads to the largest latency and, for that purpose, we have to consider two corner cases: the turnaround happens in the rank u.a., or the turnaround happens in the last rank visited in the R-Sweep (those are corner cases because their algebraic expressions separate the occurrences of $dRR\text{-}\overline{R}$ from $dWW\text{-}\overline{R}$ inside the max{} operator). So, for instance, if the turnaround happens in the rank u.a., then $dRW\text{-}R$ runs simultaneously with $(nR-1)$ occurrences of $dRR\text{-}\overline{R}$ and $dRW\text{-}\overline{R}$. Eq. 5.14 simply reflects the aforementioned observations.

We now discuss the correctness of Eq. 5.15, which computes the *ccds* component of round i. The outcome of the computation depends on the case being considered (A or B). Regardless of the case, however, we must assume that the rank u.a. contributes with $CC_{sum}(nB-1)$. The contribution of the interfering ranks depends on the case: $CC_{sum}(nB)$ per rank (case A) or $2\cdot CC_{sum}(nB/2)$ per rank (case B). The difference in expressions is because, in comparison with case A, one dCC occurrence in case B is replaced with a rank switch.

Finally, we only briefly discuss the *switches* and *ccds* components of B_{i+1}. The *switches* component, computed according to Eq. 5.12 is similar in a symmetrical fashion to the *switches* component of B_i. The *ccds* component, computed according to Eq. 5.13 is also similar to its B_i counterpart. There is, however, an important difference: it contains an extra *dCC-RG*, which refers to the latency between the last CAS executed in round i and the first CAS executed in round $i+1$. □

The worst-case latency of a *write* command is similar in a symmetrical fashion to the one of a *read*. Consequently, we simply stated Theorem 2 and provide no further discussion about it.

Theorem 2. *The worst-case latency of a* write *command is calculated with Eq. 5.9.*

$$L^W = B^i + B^{i+1} \tag{5.9}$$

where:

$$B^i = \max \begin{cases} ccds_i(CaseA) + switches_i(CaseA) \\ ccds_i(CaseB) + switches_i(CaseB) \end{cases} \tag{5.10}$$

$$B^{i+1} = \max \begin{cases} ccds_{i+1}(CaseA) + switches_{i+1}(CaseA) \\ ccds_{i+1}(CaseB) + switches_{i+1}(CaseB) \end{cases} \tag{5.11}$$

$$switches_i(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \;\; dWR\text{-}RG + dRR\text{-}\overline{R} \cdot (nR-1) \\ \textbf{\textit{else then:}} \;\; \max \begin{cases} \max\{(dWW\text{-}\overline{R} \cdot (nR-1) + dWR\text{-}\overline{R}), dWR\text{-}RG\} \;\; + dRR\text{-}\overline{R} \cdot (nR-1) \\ dWW\text{-}\overline{R} \cdot (nR-1) + \max\{(dWR\text{-}\overline{R} + dRR\text{-}\overline{R} \cdot (nR-1)), dWR\text{-}RG\} \end{cases} \end{cases} \tag{5.12}$$

$$ccds_i(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \;\; CC_{sum}(nB-1) + dCC\text{-}RG + CC_{sum}(nB-1) \cdot (nR-1) \\ \textbf{\textit{else then:}} \;\; CC_{sum}(nB-1) + dCC\text{-}RG + 2 \cdot CC_{sum}(nB/2) \cdot (nR-1) \end{cases} \tag{5.13}$$

$$switches_{i+1}(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \;\; dRR\text{-}\overline{R} \cdot (nR-1) + dRW\text{-}R \\ \textbf{\textit{else then:}} \;\; \max \begin{cases} \max\{(dRR\text{-}\overline{R} \cdot (nR-1) + dRW\text{-}\overline{R}), dRW\text{-}R\} \;\; + dWW\text{-}\overline{R} \cdot (nR-1) \\ dRR\text{-}\overline{R} \cdot (nR-1) + \max\{(dRW\text{-}\overline{R} + dWW\text{-}\overline{R} \cdot (nR-1)), dRW\text{-}R\} \end{cases} \end{cases} \tag{5.14}$$

$$ccds_{i+1}(c) = \begin{cases} \textbf{\textit{if}} \;\; c = CaseA \;\; \textbf{\textit{then:}} \; CC_{sum}(nB-1) + CC_{sum}(nB) \cdot (nR-1) \\ \textbf{\textit{else then:}} \;\; CC_{sum}(nB-1) + 2 \cdot CC_{sum}(nB/2) \cdot (nR-1) \end{cases} \tag{5.15}$$

We discuss the worst-case latency of *activates* and *precharges*. For that purpose, we firstly state Lemma 2, which captures the effect of the lower priority of *activates* and *precharges* with regard to CAS commands (see Section 4.3.3).

Lemma 2. *Given a sequence of* n activate *or* precharge *commands that can be immediately executed without violating timing constraints and that can only postpone each other for one cycle due to command bus contention, the maximum timing interval (measured in cycles) required to execute such sequence is given by Eq. 5.16.*

$$\alpha_{PA}(n) = n + \left\lceil \frac{n}{t_{BURST} - 1} \right\rceil \tag{5.16}$$

Proof. Activate and *precharge* commands have lower priority than CAS commands. Hence, to bound the timing interval required to execute them, we have to rely on information about the minimum distance between consecutive CAS commands. More specifically, in single-rank systems, any two consecutive CAS commands must be executed at least t_{BURST} cycles apart (or by even more cycles if a data bus turnaround is required). In multi-rank systems, the same statement remains true if we consider a t_{RTRS} of 4.5 nanoseconds (see Section 2.3 and Table 2.2). Hence, in any interval of t_{BURST} cycles, at least $t_{BURST} - 1$ cycles will be free for the execution of *activates* and *precharges*. Eq. 5.16 follows the aforementioned observation. □

Using Lemma 2, we can now compute the worst-case latency of *activate* commands according to Theorem 3. For ease of comprehension, we depict an example of such latency in Fig. 5.4. Notice that for the sake of clarity, the figure considers systems with $nG = 1$ and, hence, the G argument is omitted when invoking $dAA\text{-}R\overline{B}$ (Theorem 3, however, includes it).

In the figure, the *activate* u.a. is blocked once (more would not be possible) by other *activates* in the rank u.a. (the rank containing the *cr* u.a.). More importantly, notice that after a residual latency (consequence of t_{FAW}), whenever an *activate* in the rank u.a. can be immediately executed, such *activate* is blocked by a CAS command and by *activates* (could have been *precharges*) from interfering ranks. This is possible due to the scheduling performed by the Command Bus Arbiter and the AP Arbiter and is the case in instants t_0 and t_1, highlighted in the figure.

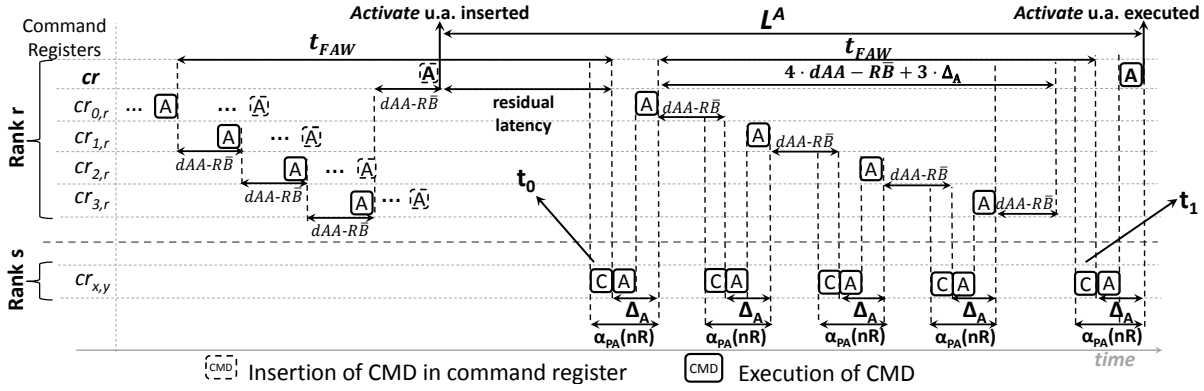


Figure 5.4: Example of the worst-case latency of an *activate* command in a hypothetical system with nB=5, nG=1 and nR=2. The t_{FAW} constraint is depicted on purpose as significantly larger than $4 \cdot dAA\text{-}R\overline{B}$ in order to highlight its effect. Moreover, the letter C represents a CAS command. Finally, the axis for $cr_{x,y}$ represents all possible registers from rank s.

Theorem 3. *The worst-case latency of a* activate *command is calculated using Eq. 5.17.*

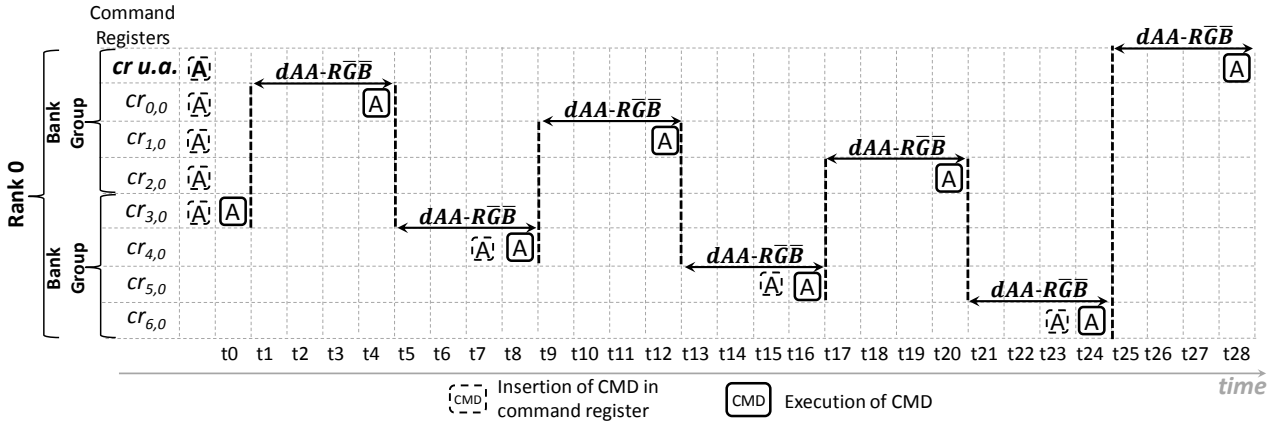
$$L^A = (t_{FAW} - (4 \cdot dAA\text{-}R\overline{GB})) + \max \begin{cases} (nB-1) \cdot dAA\text{-}R\overline{GB} + nB \cdot \Delta_A, \\ (nB-1) \cdot dAA\text{-}R\overline{GB} + nB \cdot \Delta_A + (t_{FAW} - (4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A)) \cdot K \end{cases} \tag{5.17}$$

where $\Delta_A = \alpha_{PA}(nR) - 1$ and $K = \left\lfloor \frac{(nB-1)}{4} \right\rfloor$.

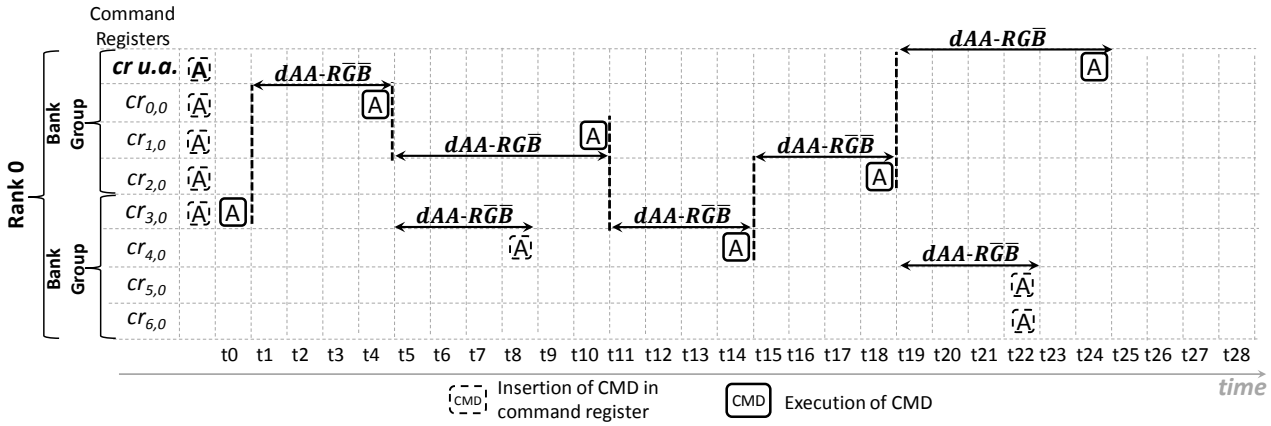
Proof. The equation that computes L^A has two main terms. The leftmost term accounts for the residual latency depicted in Fig. 5.4, which comes from the conservative assumption that 4 *activates* are executed as late as possible in the rank u.a..

The rightmost term (max operator) accounts for the remaining latencies by selecting the largest value between two expressions. The first one (upper expression) considers that the influence of t_{FAW} is hidden due to inter-rank and CAS interference. More specifically, it considers that $t_{FAW} < 4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A$ (which is not the case in Fig. 5.4). The second one (lower expression) considers exactly the opposite ($t_{FAW} > 4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A$) and simply replaces $(4 \cdot dAA\text{-}R\overline{GB} + 3 \cdot \Delta_A)$ by t_{FAW} in each of the K times in which the t_{FAW} constraint is activated.

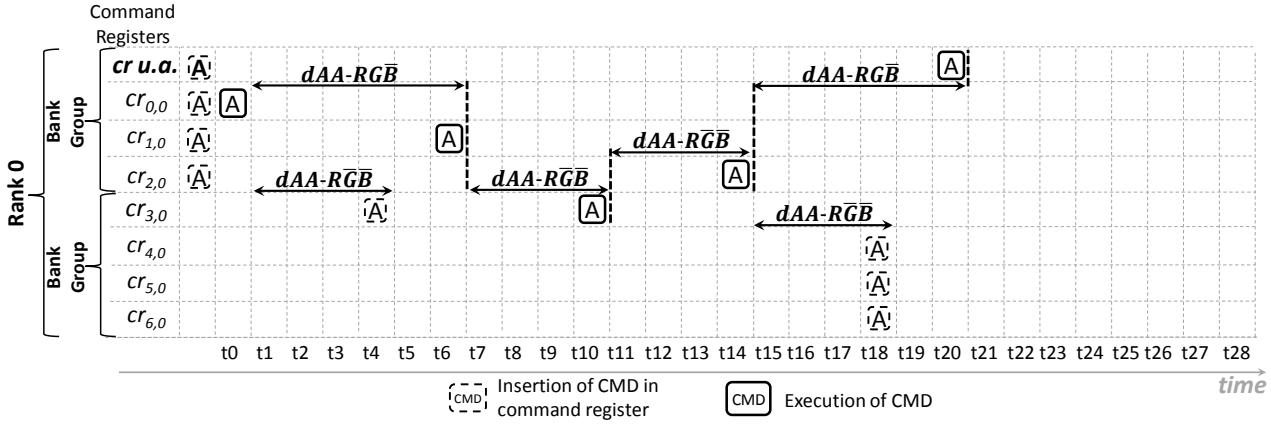
Finally, we discuss the use of the $\overline{G}$ modifier in the equations. In systems with nG=1, the G modifier is simply ignored. However, in systems with $nG \leq 2$, such modifier is important and we need to justify using $\overline{G}$ in our equations.



(a) Worst-case interference.



(b) Not the worst-case interference. Notice that in $t9$, an arbiter that purely prioritizes the oldest ready command would execute the *activate* from $cr_{4,0}$, which is not the case for an arbiter that employs *real-time aware oldest ready* arbitration. A similar observation can be made about instant $t23$.



(c) Also not the worst-case interference. Similarly to (b), the *real-time aware oldest ready* arbitration prevents execution fom the second bank group in instants $t5$ and $t19$.

Figure 5.5: Examples of non-t_{FAW} intra-rank interference that an *activate* command can suffer in a system with nR=1, nG=2 and nB=8 and in which $dAA\text{-}R\overline{GB} = 4$ and $dAA\text{-}RG\overline{B} = 6$. For the sake of the examples, t_{FAW} is assumed to be smaller than $4 \cdot dAA\text{-}R\overline{GB}$.

For the residual latency (leftmost term of Eq. 5.17), using $\overline{G}$ obviously increases the outcome of the computation, simply because $(t_{FAW} - 4 \cdot dAA\text{-}R\overline{GB})$ is larger than $(t_{FAW} - 4 \cdot dAA\text{-}RG\overline{B})$. In the expressions inside the max operator, we employ $\overline{G}$ because of the *real-time aware oldest ready* arbitration of the AP Arbiter (see Section 4.3.2). More specifically, we use $\overline{G}$ because if we consider solely the intra-rank non-t_{FAW} interference experienced by the *activate* u.a., such interference is maximized if two conditions are simultaneously satisfied:

1. firstly, the *activate* u.a. is blocked by $nB - 1$ interfering *activates* (more is not possible).
2. And secondly, all interfering command registers (in the same rank as the *activate* u.a.) suffer insertions of *activates* soon enough for a full *group interleaved* command execution to happen.

Such scenario is depicted in Fig. 5.5a. In order to prove that it indeed depicts the maximum interference, we need to observe what happens if the interfering command registers suffer late insertions of *activates*, which in turn prevent a full *group interleaved* execution pattern. This is depicted in Figs. 5.5b and 5.5c.

In order to continue our proof, let us focus on Fig. 5.5b, in which the latency of the *activate* u.a. is larger than the one in Fig. 5.5c. More specifically, the *activate* u.a. suffers a blocking that amounts to $\left(\frac{nB/nG}{2}\right) \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + \left(\frac{nB/nG}{2} - 1\right) \cdot dAA\text{-}R\overline{GB}$. The left term of the expression comes from the observation that for every pair of command registers inside the *bank group* from the *cr u.a.*, we can observe an interference of $(dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B})$ (e.g. see the two latencies that follow the execution of the *activate* in $cr_{3,0}$ in Fig. 5.5b or the two latencies that follow the execution of the *activate* in $cr_{4,0}$ in the same figure). The right term accounts for the latencies between the execution of *activates* in the command register pairs mentioned in the explanation of the left term, e.g. the $dAA\text{-}R\overline{GB}$ between $t11$ and $t14$ in Fig. 5.5b. Notice that, regardless of the number of *bank groups* in the system, a larger blocking would only be possible if the *activates* outside the *bank group* of the *cr* u.a. were inserted earlier (scenario from Fig. 5.5a).

This leads us to the last part of our proof. More specifically, let us assume that the scenario depicted in Fig. 5.5b leads to worst-case interference in a system in which the number of banks per *bank group* is equal to 4 ($nB/nG = 4$), which is the case for DDR4 systems. This means that such interference would be larger than the one depicted in Fig. 5.5a and leads to the following inequality:

$$\left(\frac{nB/nG}{2}\right) \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + \left(\frac{nB/nG}{2} - 1\right) \cdot dAA\text{-}R\overline{GB} > (nB - 1) \cdot dAA\text{-}R\overline{GB}$$

$$\left(\frac{4}{2}\right) \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + \left(\frac{4}{2} - 1\right) \cdot dAA\text{-}R\overline{GB} > (nB - 1) \cdot dAA\text{-}R\overline{GB}$$

$$2 \cdot (dAA\text{-}R\overline{GB} + dAA\text{-}RG\overline{B}) + 1 \cdot dAA\text{-}R\overline{GB} > (nB - 1) \cdot dAA\text{-}R\overline{GB}$$

$$2 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} + 1 \cdot dAA\text{-}R\overline{GB} > (nB - 1) \cdot dAA\text{-}R\overline{GB}$$

$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > (nB - 1) \cdot dAA\text{-}R\overline{GB}$$

In systems with 8 banks per rank, we can further develop the expressions until we reach a false statement:

$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > (8-1) \cdot dAA\text{-}R\overline{GB}$$
$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > 7 \cdot dAA\text{-}R\overline{GB}$$
$$2 \cdot dAA\text{-}RG\overline{B} > 4 \cdot dAA\text{-}R\overline{GB}$$
$$dAA\text{-}RG\overline{B} > 2 \cdot dAA\text{-}R\overline{GB} \quad \textbf{False!!!}$$

(Notice that $dAA\text{-}RG\overline{B}$ is indeed larger than $dAA\text{-}R\overline{GB}$, but never twice larger, as claimed by the last line of the inequation.)

Similarly, in systems with with 16 banks per rank, we also develop the expressions until we reach a false statement:

$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > (16-1) \cdot dAA\text{-}R\overline{GB}$$
$$3 \cdot dAA\text{-}R\overline{GB} + 2 \cdot dAA\text{-}RG\overline{B} > 15 \cdot dAA\text{-}R\overline{GB}$$
$$2 \cdot dAA\text{-}RG\overline{B} > 12 \cdot dAA\text{-}R\overline{GB}$$
$$dAA\text{-}RG\overline{B} > 6 \cdot dAA\text{-}R\overline{GB} \quad \textbf{False!!!}$$

Consequently, we can affirm that the worst-case interference happens in the scenario from Fig. 5.5b. Finally, we highlight that in future systems, in which the number of banks per *bank group* might be larger than 4, a proof can be achieved employing a similar strategy. This concludes our proof. □

We now compute the worst-case latency of *precharge* commands with Theorem 4.

Theorem 4. *The worst-case latency of a* precharge *command is calculated using Eq. 5.18.*

$$L^P = \alpha_{PA}(nB \cdot nR) \tag{5.18}$$

Proof. Precharge commands can be executed back-to-back regardless of the rank. Moreover, a *precharge* can be blocked at most once by *precharge* or *activate* commands in interfering banks. Eq. 5.18 reflects the aforementioned observations. Finally, notice that we employ $nB \cdot nR$ (instead of $nB \cdot nR - 1$) as an argument to the $\alpha_{PA}(n)$ function, as we also account for one cycle required to execute the *precharge* u.a.. □

5.3 Worst-case Latency of a Task

The worst-case cumulative SDRAM latency of a task (L_{Task}^{SDRAM}) refers to the the maximum amount of time that a task spends idle while waiting for its memory requests to be served. In order to compute it, we firstly compute the worst-case latency of SDRAM requests using the worst-case command latencies from the previous subsection.

The worst-case latency of a SDRAM request refers to the maximum time interval between its arrival at the SDRAM controller and the end of the corresponding data transfer. Such latency depends on three factors: the worst-case command latencies required to serve the request, the corresponding intra-bank constraints and the time interval required for the corresponding data transfer to happen. In total, there are four types of request we must consider: read miss (RM), read hit (RH), write miss (WM) and write hit (WH). The words miss and hit are not related to cache and instead refer to whether the request u.a. targets a row currently present in the corresponding row buffer or not. If that is the case, only a CAS command is required. If that is not the case, than a P-A-CAS command sequence is required.

That being said, we state Theorem 5.

Theorem 5. *The worst-case latency of a* Read Miss (RH)*, a* Read Hit (RH)*, a* Write Miss (WM) *and a* Write Hit (WH) *requests are given by Eqs. 5.19, 5.20, 5.21 and 5.22 respectively.*

$$L_{Req}^{RM} = t_{Residual} + L^P + L^A + L^R + dPA\text{-}RGB - 1 + dAR\text{-}RGB - 1 + dRD + t_{BURST} \tag{5.19}$$

$$L_{Req}^{RH} = L^R + dRD + t_{BURST} \tag{5.20}$$

$$L_{Req}^{WM} = t_{Residual} + L^P + L^A + L^W + dPA\text{-}RGB - 1 + dAW\text{-}RGB - 1 + dWD + t_{BURST} \tag{5.21}$$

$$L_{Req}^{WH} = L^W + dWD + t_{BURST} \tag{5.22}$$

where:

$$t_{Residual} = \begin{cases} \max\{(dRP\text{-}RGB - 1 - (dRD + t_{BURST})), 0\} & \text{if prev. was RH} \\ \max\{(dAP\text{-}RGB - 1 - (dAR\text{-}RGB + dRD + t_{BURST})), dRP\text{-}RGB - 1 - (dRD + t_{BURST}), 0\} & \text{if prev. was RM} \\ \max\{(dWP\text{-}RGB - 1 - (dWD + t_{BURST})), 0\} & \text{if prev. was WH} \\ \max\{(dAP\text{-}RGB - 1 - (dAW\text{-}RGB + dWD + t_{BURST})), dWP\text{-}RGB - 1 - (dWD + t_{BURST}), 0\} & \text{if prev. was WM} \\ 0 & \text{if prev. None} \end{cases} \tag{5.23}$$

Proof. In order to aid our proof, we depict the latencies that contribute to the worst-case latency of a RM request in Fig. 5.6. (The case for WM is similar, and the cases for RH and WM would simply contain no *activate* and *precharge* commands). Notice that between the execution of a command and the insertion of the next command into the command register u.a., the corresponding intra-bank latencies must be respected. For instance, after the execution of a *precharge*, the corresponding bank scheduler has to wait $dPA\text{-}RGB - 1$ cycles before inserting an *activate* into the command register u.a.. The -1 term is explained in Section 4.2.

Eqs. 5.19, 5.20, 5.21 and 5.22 simply sum worst-case command latencies, the intra-bank latencies and the data transfer duration. However, one characteristic of the equations for RM and WM requests demands a clarification. More specifically, the $t_{Residual}$ term. The $t_{Residual}$ latency is a consequence of intra-bank timing constraints ($dAP\text{-}RGB$, $dRP\text{-}RGB$ and $dWP\text{-}RGB$) that limit how fast a *precharge* can be inserted into the command register u.a.. Such latency depends on the type of the request that preceded the request u.a. and, in order to compute it, we conservatively assume that the request u.a. arrives exactly after the previous request has been served, as depicted in Fig. 5.6

□

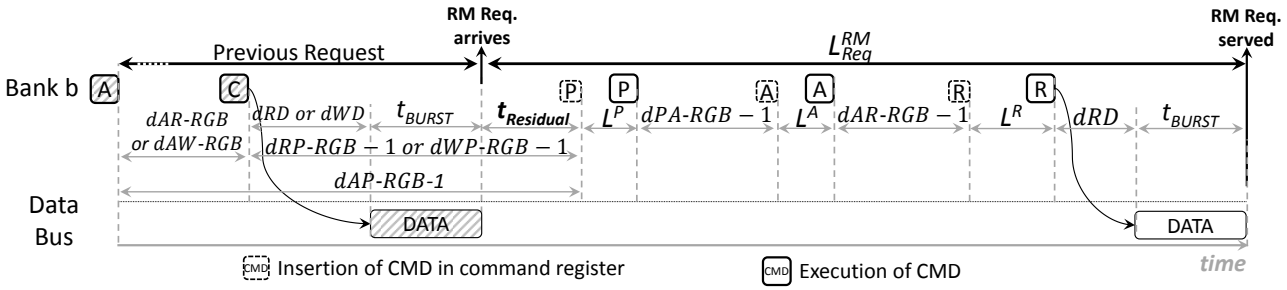


Figure 5.6: Latency decomposition of a RM request. The figure is not drawn to scale and the employed proportions are chosen solely to properly fit the latency labels. Moreover, the letter C refers to a CAS command.

We now discuss the worst-case SDRAM latency of a task (L_{Task}^{SDRAM}). For that purpose, we assume that the number and the types of requests made by a task is extracted from a trace. This eases our computation of L_{Task}^{SDRAM}, as we can always select the appropriate value of $t_{Residual}$ for RM and WM requests. (If such assumption cannot be made, a static timing analysis tool [22] can be employed to extract the maximum number and type of requests that a task can perform.

Moreover, a worst-case bound on the sum of all $t_{Residual}$ latencies must be derived. We refer the interested reader to [6].)

Theorem 6. *The worst-case SDRAM Latency of a task whose SDRAM request trace is available is computed using Algorithm 3.*

Proof. The algorithm simply sums the latency of every request in the trace, which comes directly from our assumption of a timing compositional processor that stalls at every request (see Section 5.1). □

Algorithm 3 Computes L_{Task}^{SDRAM}

```
1: // Inputs: N (number of requests) and request_trace (a trace with N requests)
2: function Compute_Cumulative_WC_Latency(N, request_trace[N])
3:     L_Task^SDRAM ← 0 ;
4:     previous_request ← None ;
5:     current_request ← None ;
6:     for index ← 0; index < N; index ← index + 1 do
7:         current_request ← request_trace[index] ;
8:         t_Residual ← Get_tResidual(previous_request) ;
9:         L_Task^SDRAM ← L_Task^SDRAM + Get_Request_Latency(t_Residual, current_request) ;
10:        previous_request ← current_request ;
11:    return L_Task^SDRAM ;
```

6 Evaluation

In this section, we present our evaluation (which is based on SDRAM request traces from applications). We firstly discuss the traces and the experimental setup. Then, we compare our approach with another *open-row* real-time controller. Finally, we evaluate worst-case performance trends in SDRAM modules from different DDR generations and speed bins.

6.1 Application Request Traces and Experimental Setup

In order to collect traces, applications are executed in Gem5 [23] with a 1.1 GHz scalar ARM processor with 64-kb of L1 cache (split evenly between instructions and data). The cache line size is 64 bytes (which matches the access granularity of a SDRAM module with a 64-bit data bus). Moreover, the cache employs write-back policy. As for applications, we selected 5 out of a set of 16 applications from Mibench [24] and EEMBC [25]. The applications and their profiles (proportion of request types) are depicted in Fig. 6.1. Notice that the selected applications, which are highlighted in the Fig. 6.1, have very different profiles (e.g. *gsm* has a very high number of requests that hit at the row buffer, which is not the case for *cacheb01*).

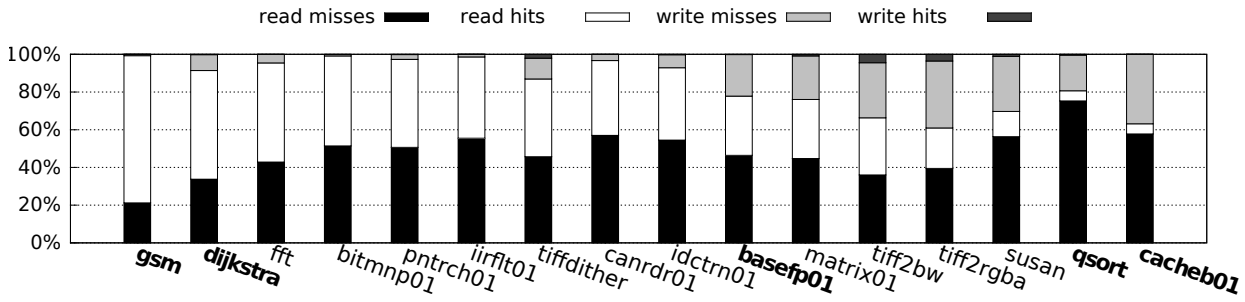


Figure 6.1: Percentage of each request type for applications. The words *hits* and *misses* refer to the row buffer of an SDRAM bank, not to L1-cache. The applications used for our evaluation are highlighted.

Because the number of requests of each trace varied drastically (from a couple hundred thousands to millions), we *summarize* the traces. More specifically, we generate artificial traces, each containing 10000 requests, but respecting the proportions depicted in Fig. 6.1. The summarization (which is formally described with Algorithm 4) allows us to compare L_{Task}^{SDRAM} over different applications using absolute values and, hence, observe how the request types affect overall latency. Moreover, it drastically reduces simulation times.

We now discuss the experimental setup. Our evaluation consists in comparing analytical bounds of applications with results obtained with cycle-accurate controller simulators. For that purpose, we assume SDRAM modules with 64-bit data buses (and that each bank of the module has an 8 KB row buffer). Moreover, we give the application under analysis exclusive access to one of the banks. The other banks are occupied by interference generators that trigger back-to-back requests. The generators are programmed so that each request has a 40%, 40%, 10% and 10% probability of being a read hit, write hit, read miss and write miss, respectively. We choose such settings because the complexity of the controllers investigated in this technical report mainly regards the scheduling of CAS commands (and not *activates* and *precharges*). Moreover, the high ratio of

Algorithm 4 Summarizes a trace

```
1: // Inputs: N (number of requests of original trace), request_trace (a trace with N requests),
2: //          summarized_N (number of requests in the summarized trace)
3: // Output: summarized_trace[summarized_N]
4: function Summarize_Trace(N, request_trace[N], summarized_N)
5:     summarized_trace ← new Request_Trace[summarized_N] ;
6:
7:     n_of_rm ← Count_Number_Of_Read_Misses(request_trace) ;
8:     n_of_wm ← Count_Number_Of_Write_Misses(request_trace);
9:     n_of_rh ← Count_Number_Of_Read_Hits(request_trace) ;
10:    n_of_wh ← Count_Number_Of_Write_Hits(request_trace) ;
11:
12:    // Computes probability of each request appearing in trace
13:    prob_rm ← n_of_rm / N;
14:    prob_wm ← n_of_wm / N;
15:    prob_rh ← n_of_rh / N;
16:    prob_wh ← n_of_wh / N;
17:
18:    for index ← 0; index < summarized_N; index ← index + 1 do
19:        new_request ← Generate_Request(prob_rm, prob_wm, prob_rh, prob_wh) ;
20:        summarized_trace[index] ← new_request ;
21:    return summarized_trace ;
22:
23: function Generate_Request(prob_rm, prob_wm, prob_rh, prob_wh)
24:    range_rm ← prob_rm ;
25:    range_wm ← prob_rm + prob_wm ;
26:    range_rh ← prob_rm + prob_wm + prob_rh ;
27:    range_wh ← prob_rm + prob_wm + prob_rh + prob_wh ;
28:
29:    aux ← Generate_Random_Float_Between_0_and_1( );
30:
31:    if aux < range_rm then
32:        new_request ← new ReadMissRequest();
33:    else if aux < range_wm then
34:        new_request ← new WriteMissRequest();
35:    else if aux < range_rh then
36:        new_request ← new ReadHitRequest();
37:    else
38:        new_request ← new WriteHitRequest();
39:    return new_request
```

writes is meant to cause frequent turnarounds.

Finally, in order to uniquely identify the SDRAM modules investigated in this report, we use a string with the DDR generation, model and a suffix that describes its structure. For instance, DDR4-2400U-2r,2g,8b refers to a dual-rank module built using DDR4-2400U devices with 8 banks divided into 2 *bank groups* (nR=2,nG=2,nB=8).

6.2 Comparison with Related Work

In this subsection, we compare our SDRAM controller with the Waterloo controller [6, 7, 10], which, as discussed in Section 2.4, does not employing CAS command reordering to minimize turnarounds and rank switches. We make four important highlights about our comparison: firstly, in order to generate analytical bounds for the Waterloo controller, we also assume that the order of the requests made by a real-time task is known, which allows us to accurately compute $t_{Residual}$. Secondly, our comparison is made in terms of interfering banks. Consequently, regardless of the controller, the cumulative SDRAM latency of an application is always better in single-rank setups because they contain only 7 interfering banks (while dual- and quad-rank setups contain 15 and 31, respectively). Thirdly, for quad-rank modules, we assume $t_{RTRS} = 9$ nanoseconds (twice larger as the t_{RTRS} for dual-rank modules, which is discussed in Section 2.3) because there are two extra ranks connected to the data bus. And finally, please notice that our comparison is limited to DDR3 modules, as there is no trivial way to properly extend the analysis from [6, 7, 10] in order to account for the DDR4 *bank groups* feature.

For our comparison, we consider single-, dual- and quad-rank modules built using DDR3-1600K and DDR3-2133N devices with nB=8 banks. We choose different speed bins because they help to highlight one key difference between our controller and the Waterloo controller: in our controller, each CAS can be blocked twice by CAS commands in interfering banks due to the command reordering performed by read/write bundling (see Figs 5.2a and 5.2b). In the Waterloo controller, which employs no CAS reordering, each CAS can only be blocked once by each interfering bank. From the analytical perspective, in order for the reordering to pay off, the overhead for bus turnarounds and rank switches must be large and, consequently, the advantage of our approach is better displayed in high-speed modules.

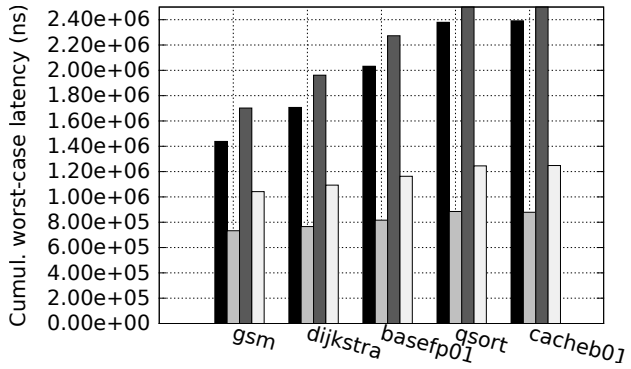
That being said, we present the results or our comparison in Figs. 6.2 and 6.3. In the figures, the smaller the bar, the better the result.

We summarize the results with five main observations:

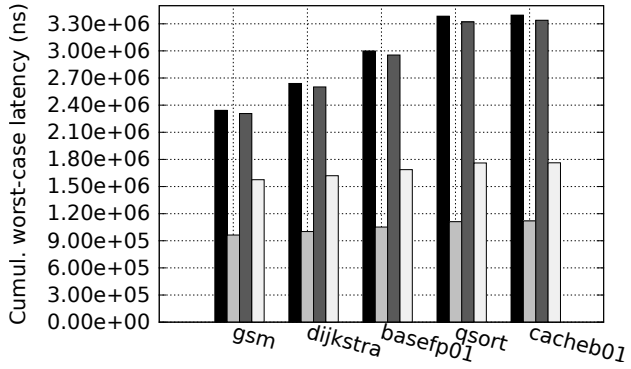
1. In terms of analytical bounds and in comparison with the Waterloo controller, the advantage of our controller is larger for single-rank setups (as a matter of fact, for the DDR3-1600K-2r,1g,8b module, our bounds are slightly worse than the ones provided by the Waterloo controller). This is because in multi-rank setups, the Waterloo controller enforces that turnarounds happen concurrently with rank switches, while in our controller, each CAS command can experience two full data bus turnarounds (see round i and round $i+1$ in Fig. 5.2a).

2. In terms of analytical bounds in multi-rank modules and in comparison with the Waterloo controller, our controller provides better bounds in quad-rank (rather than in dual-rank) modules. This is because of our assumption that $t_{RTRS} = 9$ nanoseconds in quad-rank modules.

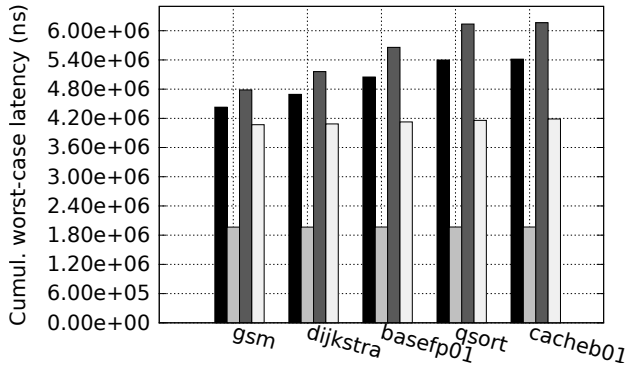
3. In terms of experimental performance, our controller performs better than the Waterloo controller. The statement remains true even if we consider a DDR3-1600K-2r,1g,8b module, for which our analytical bounds are slightly worse. The reason that explains this behavior



(a) For a DDR3-1600K-1r,1g,8b module.

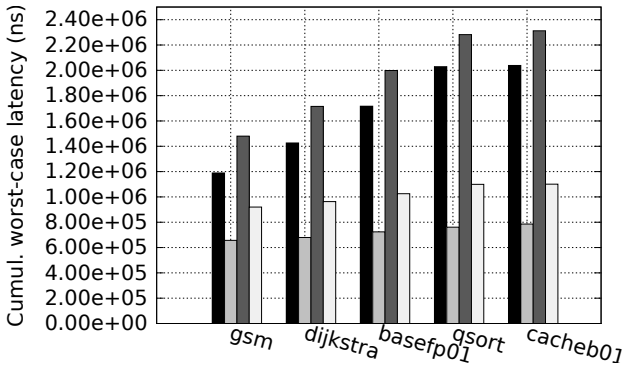


(b) For a DDR3-1600K-2r,1g,8b module.

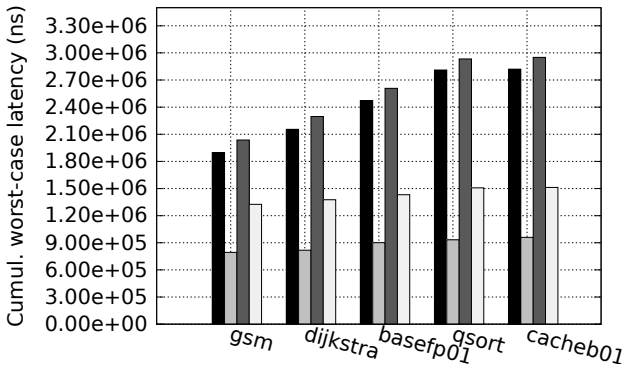


(c) For a DDR3-1600K-4r,1g,8b module.

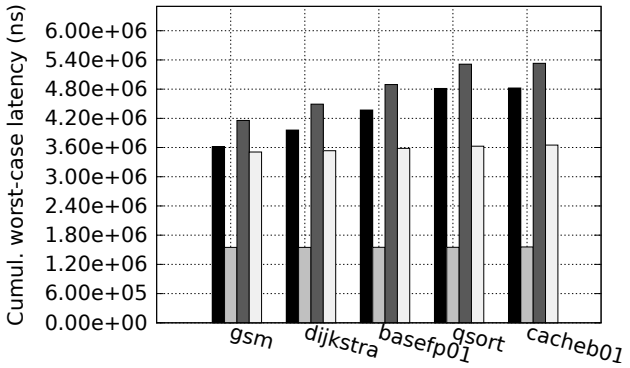
Figure 6.2: Comparison of cumulative worst-case latencies between our controller and the Waterloo controller [6, 7, 10] for single-, dual- and quad-rank modules built using DDR3-1600K devices. Notice that each figure employs a different scale in the y-axis.



(a) For a DDR3-2133N-1r,1g,8b module.



(b) For a DDR3-2133N-2r,1g,8b module.



(c) For a DDR3-2133N-4r,1g,8b module.

Figure 6.3: Comparison of cumulative worst-case latencies between our controller and the Waterloo controller [6, 7, 10] for single-, dual- and quad-rank modules built using DDR3-2133N devices. Notice that each figure employs a different scale in the y-axis.

(worse analytical bounds and better experimental performance) is that although the analytical bounds of our controller must assume that a CAS command is blocked twice by each interfering bank (see Figs 5.2a and 5.2b), CAS commands are mostly executed in the scheduling round being performed while they were inserted. More specifically, in most of the time, they are only blocked once by each interfering bank.

4. Also in terms of experimental performance, notice that the gap between ours and the Waterloo controller increases with the number of ranks. For instance, for quad-rank modules, the cumulative latency of applications running in our controller is less than half than what is observed in the Waterloo controller. This is because the Waterloo controller constantly performs rank switches. Consequently, as we assume $t_{RTRS} = 9$ nanoseconds for the quad-rank modules, i.e. twice larger than for dual-rank modules, the performance of the Waterloo controller drops significantly. (Here we highlight that although experimental performance is irrelevant for hard real-time tasks as long as it is not worse than the analytical bounds, providing high performance is a desirable requirement in mixed critical systems, in which some of the banks are assigned for soft real-time applications [9, 7].)

5. And finally, regardless of the controller and considering row buffer hit ratio as a parameter, applications with high row buffer hit ratio, e.g. *gsm*, have better analytical bounds than the ones with low row buffer hit ratio, e.g. *cacheb01*. In terms of experimental cumulative latencies, applications with high row buffer hit ratio also perform better than their low row buffer hit ratio counterparts. However, the difference between them is small when compared with the difference observed in analytical bounds (the difference in quad-rank modules is barely noticeable). This is because during the simulation, intra-bank latencies (time required to precharge and activate a row) tended to overlap with inter-bank interference, a behavior that cannot be assumed by the timing analysis.

6.3 Worst-Case Performance Trends Across Different DDR Devices and Generations

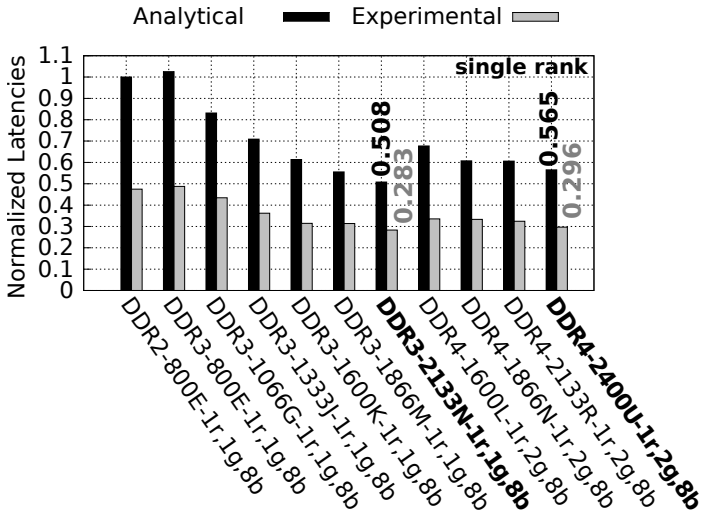
In this subsection, we compare the worst-case analytical and observed bounds provided by our SDRAM controller over a wide range of SDRAM modules. As we previously discussed, this article is to our knowledge the first (and at the moment the only) to address generation-independent scheduling from the perspective of *open-row* real-time SDRAM controllers. Hence, in this subsection, we do not compare ourselves with related work.

We select two applications for our evaluation: *gsm*, which contains a high number of row buffer hits, and *cacheb01*, which contains a low number of row buffer hits. We compute analytical bounds for the applications and perform experimental simulations as described in Section 6.1 considering firstly systems with 8 banks and then systems with 16 banks. We depict the results for systems with 8 banks in Fig. 6.4 and for systems with 16 banks in Fig. 6.5.

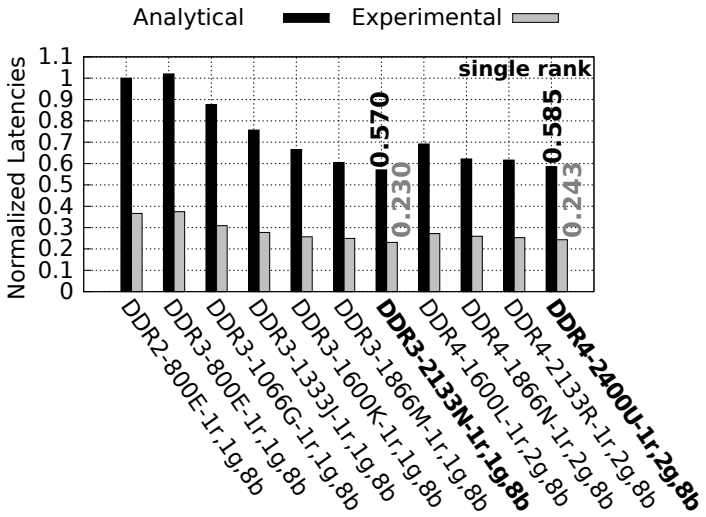
We firstly highlight the observed trends (for both 8- and 16-bank systems) with three observations:

1. The cumulative SDRAM latencies observed during the experimental simulation are always smaller than the analytical bounds.

2. Because CAS commands cannot always be executed in a *group interleaved* pattern in DDR4 systems (see *reads* in rank s in Fig. 4.2b), DDR4 SDRAMs perform slightly worse than DDR3 SDRAMs both from the perspective of worst-case bounds and from the perspective of experimental simulation results. Such statement remains true even if we compare devices with different operating frequencies, e.g. DDR3-2133N and DDR4-2400U.

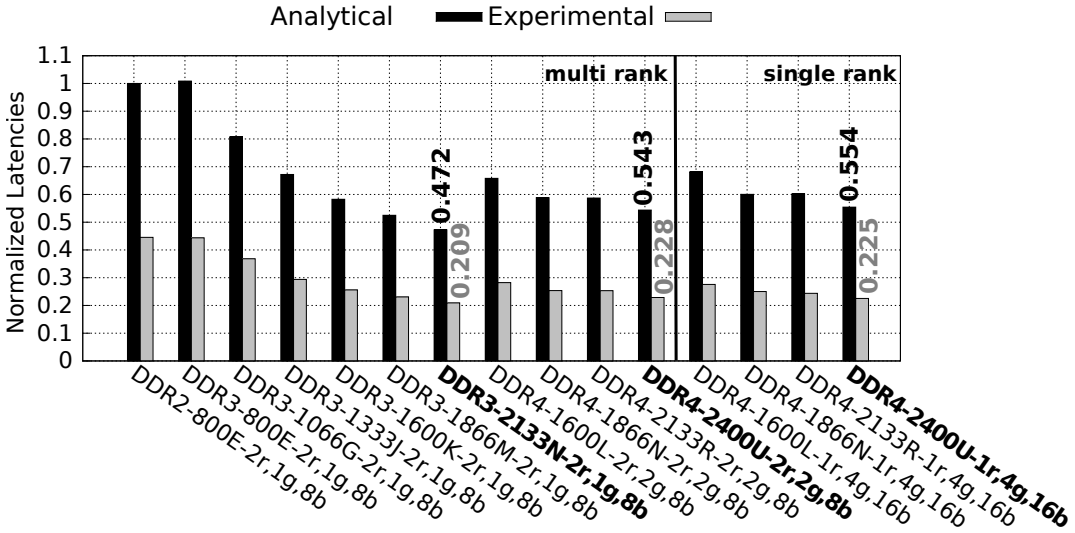


(a) *Gsm.*

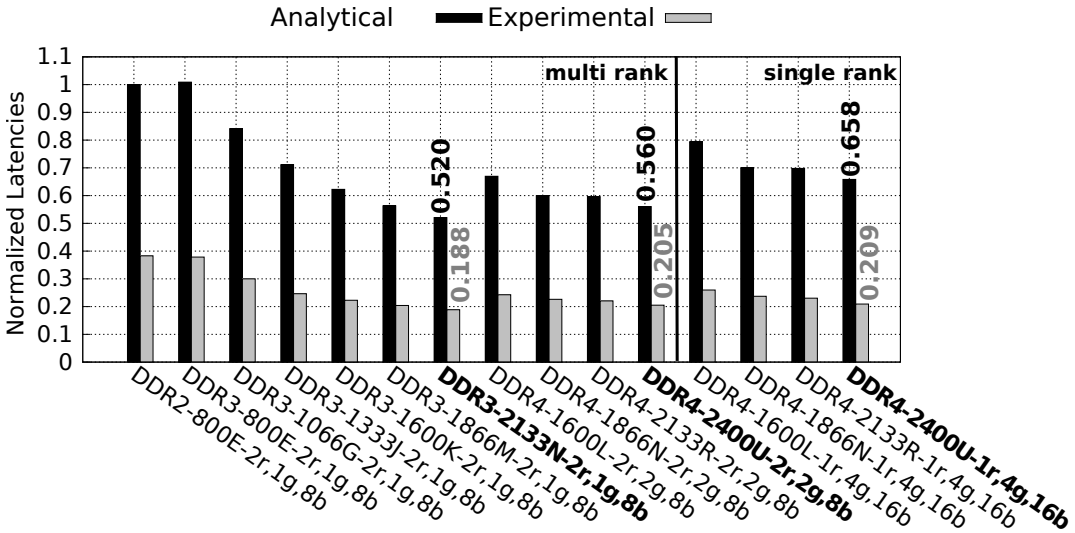


(b) *Cacheb01.*

Figure 6.4: Worst-case cumulative latency of *gsm* and *cacheb01* applications over different modules with 8 banks. In (a) and in (b), results are normalized to the one obtained for the leftmost module (DDR2-800E-1r,1g,8b). Moreover, the smallest analytical and experimental latencies obtained for DDR3 and DDR4 are **highlighted**.



(a) *Gsm.*



(b) *Cacheb01.*

Figure 6.5: Worst-case cumulative latency of *gsm* and *cacheb01* applications over different modules with 16 banks. In (a) and (b), all results are normalized to the one obtained for the leftmost module, i.e. DDR2-800E-2r,1g,8b. Moreover, the smallest analytical and experimental latencies obtained for DDR3 and DDR4 are **highlighted**.

3. And finally, the worst-case bounds for *gsm*, which displays high row buffer hit ratio, are tighter (closer to the results obtained experimentally) than the ones for *cacheb01*, which displays low row buffer hit ratio. This is partially because only 20% of the requests made by interference generators demand *precharges* and *activates*. But mainly because during the experimental simulation, intra-bank latencies tend to overlap with inter-bank interference (a behavior that cannot be assumed by the timing analysis). For the interested reader, [20] discusses architectural support to enforce that such overlap occurs.

We now discuss exclusively the 16 bank system results. Such systems can only be implemented as a dual-rank setup if DDR2 or DDR3 are used, but can be implemented either as a single- or a dual-rank setup if DDR4 is used. With that respect, we make two further observations:

1. For *gsm*, the difference between the worst-case bounds obtained with single- and dual-rank DDR4 is small. This is because the largest source of inter-bank interference regards the data bus, as most requests only require CAS commands due to the large row buffer hit ratio of the application (see Fig. 6.1).

2. For *cacheb01*, however, the worst-case bounds are significantly worse for the single-rank DDR4. This is because the application displays a low row buffer locality and, hence, several of its requests demand *activate* commands, which in turn have better worst-case latency in the dual-rank setup, as less occurrences of t_{FAW} must be accounted for (see Theorem 3). Nevertheless, from the perspective of experimental results, no large difference between the single- and multi-rank setups were observed (again because intra-bank latencies tend to overlap with inter-bank interference).

7 Conclusion and Future Work

In this technical report, we propose a generation-independent *open-row* real-time SDRAM controller. Moreover, we compare our controller with the related work and evaluate the worst-case performance trends for different speed bins and module configurations from DDR2, DDR3 and DDR4 SDRAMs. Our evaluation shows that due to increased complexity in the protocol to transfer data into/from DDR4 SDRAMs, they perform worse than DDR3 from the real-time perspective. However, DDR4 SDRAMs employ a lower operating voltage. This can potentially lead to significant power savings and poses an interesting direction for future work.

Bibliography

[1] Mellanox Technologies, "Tile-gx36 processor - product brief," 2015-2016. [Online]. Available: http://www. mellanox. com

[2] ——, "Tile-gx72 processor - product brief," 2015-2016. [Online]. Available: http://www. mellanox. com

[3] M. Paolieri, E. Quiñones, and F. J. Cazorla, "Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 64:1–64:26, Mar. 2013.

[4] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," in *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM Press New York, NY, USA, Sep. 2007, pp. 251–256.

[5] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: bank privatization for predictability and temporal isolation," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '11. New York, NY, USA: ACM, 2011, pp. 99–108.

[6] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, Dec 2013, pp. 372–383.

[7] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row dram controller for time-predictable systems," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 27–38.

[8] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *Real-Time Systems Symposium (RTSS), 2015 IEEE*, Dec 2015, pp. 53–64.

[9] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing DRAM rank switching overhead for improved timing bounds and performance," in *Euromicro Conference on Real-Time Systems (ECRTS) 2016*, July 2016.

[10] Z. P. Wu, R. Pellizzoni, and D. Guo, "A composable worst case latency analysis for multi-rank dram devices under open row policy," *Real-Time Systems*, pp. 1–47, 2016.

[11] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens, "Power/performance trade-offs in real-time sdram command scheduling," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1882–1895, June 2016.

[12] *JESD79-2F: DDR2 SDRAM Specification*, JEDEC, Arlington, Va, USA, Nov. 2009.

[13] *JESD79-3F: DDR3 SDRAM Specification*, JEDEC, Arlington, Va, USA, Jul. 2012.

[14] *JESD79-4: DDR4 SDRAM Specification*, JEDEC, Arlington, Va, USA, Sep. 2012.

[15] W. Wang, T. Dey, J. Davidson, and M. Soffa, "Dramon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 380–391.

[16] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, Dec 2009.

[17] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2015, pp. 307–316.

[18] Y. Li, B. Akesson, K. Lampka, and K. Goossens, "Modeling and verification of dynamic command scheduling for real-time memory controllers," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–12.

[19] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2015, pp. 317–326.

[20] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems (to appear)," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2017.

[21] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 7, pp. 966–978, July 2009.

[22] R. Bourgade, C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for wcet estimation," in *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, 2008.

[23] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[24] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[25] E. M. B. Consortium *et al.*, "Eembc benchmark suite," 2008.